Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина
Институт естественных наук и математики

С. И. СОЛОДУШКИН
И. Ф. ЮМАНОВА

РАЗРАБОТКА ПРОГРАММНЫХ КОМПЛЕКСОВ НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИМЕНИ ПЕРВОГО ПРЕЗИДЕНТА РОССИИ Б. Н. ЕЛЬЦИНА

С. И. Солодушкин, И. Ф. Юманова

РАЗРАБОТКА ПРОГРАММНЫХ КОМПЛЕКСОВ НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

Рекомендовано методическим советом
Уральского федерального университета
для студентов вуза, обучающихся по направлениям подготовки
02.03.03 «Математическое обеспечение и администрирование
информационных систем», 02.03.01 «Математика и компьютерные науки»,
02.03.02 «Фундаментальная информатика и информационные технологии»,
по специальности 10.05.01 «Компьютерная безопасность»

Екатеринбург
Издательство Уральского университета
2020

УДК 004.4(075.8)
ББК 32.973.2я73
С60

Под общей редакцией
В. Г. Пименова

Рецензенты:
кафедра прикладной математики и технической графики
Уральского государственного архитектурно-художественного
университета (заведующий кафедрой
доктор физико-математических наук, профессор *С. С. Титов*);
А. Б. Ложников, кандидат физико-математических наук, доцент,
старший научный сотрудник отдела дифференциальных уравнений
Института математики и механики УрО РАН

Солодушкин, С. И.

С60 Разработка программных комплексов на языке JavaScript : учебное пособие / С. И. Солодушкин, И. Ф. Юманова ; под общ. ред. В. Г. Пименова ; Министерство науки и высшего образования Российской Федерации, Уральский федеральный университет. — Екатеринбург : Изд-во Урал. ун-та, 2020. — 132 с. : ил. — 100 экз. — ISBN 978-5-7996-3034-8. — Текст : непосредственный.

ISBN 978-5-7996-3034-8

Рассматриваются вопросы практического использования языка JavaScript для разработки веб-сервисов. Основное внимание уделяется подробному разбору сложных теоретических вопросов прототипного наследования, политик безопасности. В качестве средств разработки используются платформа Node.js и фреймворк Express.

Для студентов, специализирующихся в области прикладной информатики, компьютерных наук и занимающихся разработкой программного обеспечения.

УДК 004.4(075.8)
ББК 32.973.2я73

ISBN 978-5-7996-3034-8

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Учебное пособие «Разработка программных комплексов на языке JavaScript» написано авторами на основе опыта чтения курса «Web и DHTML» в Уральском федеральном университете. Цель учебного пособия — изучение объектно-ориентированного программирования на JavaScript и прототипного наследования, Same Origin Policy и методов релаксации тех жестких ограничений, которые данная политика накладывает, а также знакомство с фреймворком Express.

Учебное пособие призвано помочь студентам в освоении курса «Web и DHTML» и отражает его структуру. Пособие разбито на главы. Каждая глава соответствует рассматриваемой на занятиях теме и содержит необходимые теоретические сведения, примеры, всесторонне иллюстрирующие теорию, и иногда листинги программ. Кроме того, в конце глав приводятся вопросы для самоконтроля.

При подготовке учебного пособия авторы в основном обращались к первоисточникам, т. е. к официальным стандартам и документации.

Работа выполнена в рамках исследований Уральского математического центра.

Глава 1
ОСНОВНЫЕ ПОНЯТИЯ И ИСТОРИЯ ЯЗЫКА JAVASCRIPT

1.1. Краткая история появления языка JavaScript: причины и предпосылки

Необходимо понимать, что языки программирования создаются в контексте определенных исторических реалий, и нужна достаточно сильная мотивация для разработки нового языка. Разработчики языка ставят перед собой вполне конкретные цели, а это, в свою очередь, определяет возможности языка, реализуемые в нем парадигмы. В связи с этим мы дадим историческую справку о появлении и развитии языка JavaScript, а потом охарактеризуем его в общем и целом.

Начало 1990-х гг. ознаменовано бурным развитием Интернета и особенно World Wide Web. Разработчики браузеров (на тот момент было два основных браузера — Internet Explorer и Netscape Navigator) старались сделать свои продукты более продвинутыми, реализовать в них поддержку новых возможностей и тем самым вытеснить конкурентов и завоевать рынок. Выполнение в браузере программного кода было одной из таких возможностей. Что происходило в мире IT в то время?

В 1992 г. компания Nombas начала разработку скриптового языка Cmm (Си-минус-минус), который, по замыслу разработчиков, должен был стать достаточно мощным, чтобы заменить макросы, сохраняя при этом схожесть с Си, чтобы разработчикам не составило

большого труда изучить его. Главным отличием от Си была работа с памятью. В новом языке все управление памятью осуществлялось автоматически: не было необходимости создавать буферы, объявлять переменные, осуществлять преобразование типов. В остальном языки походили друг на друга: в частности, Cmm поддерживал стандартные функции и операторы Си. Cmm был переименован в ScriptEase, поскольку исходное название звучало слишком негативно, а упоминание в нем Си отпугивало людей.

На основе этого языка был создан проприетарный продукт CEnvi. В конце ноября 1995 г. Nombas разработала версию CEnvi для внедрения в веб-страницы. Страницы, которые можно было изменять с помощью скриптового языка, получили название Espresso Pages — они демонстрировали использование скриптового языка для создания игры, проверки пользовательского ввода в формы и создания анимации. Espresso Pages позиционировались как демоверсия, призванная помочь представить, что случится, если в браузер будет внедрен язык Cmm. Работали они только в 16-битовом Netscape Navigator под управлением Windows.

Таким образом, идея, что браузер должен поддерживать язык программирования, встраиваемый в HTML-код страницы, витала в IT-сообществе; язык JavaScript создавался не на пустом месте.

В апреле 1995 г. компания Netscape поставила перед Бренданом Эйхом задачу — внедрить в браузер Netscape поддержку языка программирования. Анонс JavaScript представителями Netscape и Sun состоялся накануне выпуска второй бета-версии Netscape Navigator.

В июле 1996 г. компания Microsoft выпустила аналог языка JavaScript, названный JScript. Первым браузером, поддерживающим эту реализацию, был Internet Explorer 3.0.

По инициативе компании Netscape была проведена стандартизация языка ассоциацией ECMA. Стандартизированная версия, описываемая стандартом ECMA-262, имеет название ECMAScript. Первой версии спецификации соответствовал JavaScript версии 1.1, а также язык JScript.

1.2. Тезаурус (основные понятия и связи между ними)

Прежде чем переходить к описанию языка JavaScript, необходимо дать базовые определения и установить связи между ними. Дело в том, что в учебниках и особенно в электронных ресурсах зачастую имеет место смешение и/или подмена понятий, что вызывает немалую путаницу.

В этом разделе будут указаны семантические отношения между терминами. Это позволит выявить смысл терминов не только с помощью определений, но и посредством их соотнесения с другими понятиями и их группами. В свою очередь, специальная терминология должна способствовать правильной лексической, корпоративной коммуникации (общению и взаимодействию лиц, связанных одной дисциплиной или профессией).

Язык JavaScript, спецификация — описание языка, разработанного в компании Netscape. Существует в виде формального документа, который, в частности, разъясняет существующие типы данных, зарезервированные слова языка, их синтаксис и семантику. Язык JavaScript развивался, были выпущены новые релизы, где были добавлены дополнительные языковые возможности (например, стрелочные функции, блочные области видимости и т. д.).

На JavaScript можно написать программный код, однако чтобы этот код выполнить, необходимо наличие программы-интерпретатора JavaScript-кода. Таким образом, наряду со спецификацией языка существует еще реализация языка — программа, которая умеет прочитывать код, написанный на JavaScript, и выполнять его (см. движок JavaScript).

Название JavaScript является зарегистрированной торговой маркой.

Язык JScript, спецификация — описание языка, разработанного в компании Microsoft. Далее можно повторить все, что было выше написано про JavaScript. Поскольку идея Netscape выполнять скрипты на веб-страницах оказалась привлекательной, компания Microsoft также реализовала эту идею в своих браузерах.

Слово «JavaScript» является зарегистрированной торговой маркой, а потому во избежание юридических конфликтов корпорация Microsoft должна была придумать иное название для своего языка, который почти идентичен языку JavaScript.

Отметим, что первые релизы JavaScript и Jscript состоялись до выхода документации ECMAScript.

Как JavaScript, так и Jscript на момент их публикации были «корпоративными» стандартами, т. е. не были утверждены независимыми организациями по стандартизации, такими, например, как ISO, IETF или ECMA. По инициативе компании Netscape была проведена стандартизация языка ассоциацией ECMA. Стандартизированная версия имеет название ECMAScript, описывается стандартом ECMA-262.

ECMAScript, спецификация — описание языка, возникшего на основе нескольких технологий, самыми известными из которых являются языки JavaScript и JScript. Разработка первой редакции спецификации началась в ноябре 1996 г. Принятие спецификации состоялось в июне 1997 г. Язык ECMAScript стандартизирован международной организацией ECMA в спецификации ECMA-262; эта же организация занимается поддержкой и развитием языка.

Как указано в [1], «ECMAScript is based on several originating technologies, the most well-known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with Internet Explorer 3.0». Приведем перевод этого фрагмента. Язык ECMAScript основан на нескольких ориганальных технологиях, наболее известными из которых являются JavaScript (Netscape) и JScript (Microsoft). Язык был предложен Бренданом Эйхом из компании Netscape, впервые появился в ее браузере Navigator 2.0. Этот язык входил также в состав всех последующих браузеров компании Netscape и всех браузеров компании Microsoft, начиная с Internet Explorer 3.0.

ECMAScript — это абстрактное описание языка, существующее только на бумаге. Изначально ECMAScript возник на основе JavaScript или JScript как попытка их стандартизации, и авторы стандарта стремились соответствовать реализациям. Теперь, наобо-

рот, стандарт развивается относительно самостоятельно, а авторы реализаций (наиболее известными из которых являются JavaScript и JScript) стремятся соответствовать стандарту.

Реализации JavaScript и JScript, которые также называют расширениями языка, не на 100 % совместимы со стандартом ECMAScript.

Движок JavaScript — программа, которая выполняет JavaScript-код. Может быть реализована как традиционный интерпретатор или как just in time (на лету, во время исполнения) компилятор.

Когда мы говорим о реализации стандарта ECMAScript, мы говорим, во-первых, о движке, реализующем стандарт, а во-вторых, о возможных проприетарных расширениях языка.

К компетенции движков относятся вопросы оптимизации, скорости выполнения кода, расхода оперативной памяти и т. д. Существуют реализации движков под Windows, Linux, MacOS, а спецификаций под Windows, Linux нет — спецификация одна.

Приведем примеры движков:

1) SpiderMonkey — самый первый движок JavaScript, созданный Бренданом Эйхом в Netscape Communications;

2) Rhino, разрабатываемый Mozilla Foundation движок JavaScript с открытым исходным кодом, полностью написанный на Java;

3) V8-движок, разработанный датским отделением компании Google.

Среда выполнения — программная платформа, окружение, в котором работает движок JavaScript.

Начиная писать первые программы на JavaScript, мы используем console.log или setTimeout, однако в стандарте языка ничего этого не описано. Соответственно, движок JavaScript этих функций и/или объектов не знает. Возникает вопрос: а как же это тогда работает? Дело в том, что программный код на JavaScript исполняется в среде выполнения, наиболее распространенной средой являются веб-браузер и Node.js. Среда выполнения предоставляет определенные API и объекты среды, которые могут быть использованы движком.

Как указано в документации [1], ECMAScript представляет собой объектно-ориентированный язык программирования, предназначенный для проведения вычислений с объектами и управления этими объектами в *среде выполнения*.

В соответствии с определением, приведенным в этом документе, язык ECMAScript не предназначен для использования в качестве самодостаточной вычислительной системы, и в спецификации даже отсутствуют средства для ввода внешних данных или вывода результатов вычислений. Однако предполагается, что именно среда выполнения предоставит программе на языке ECMAScript не только объекты и иные средства, описанные в спецификации, но и некоторые объекты, определяемые конкретной средой. Описание и поведение этих объектов не является предметом спецификации ECMA-262, однако необходимо отметить, что упомянутые объекты могут предоставлять определенные свойства, к которым можно обратиться из программы, написанной на ECMAScript, и определенные функции, которые можно вызвать из программы на ECMAScript.

Изначально ECMAScript задумывался как язык веб-скриптов, обеспечивающий механизм, позволяющий оживлять интернет-страницы браузеров и производить вычисления на стороне сервера в рамках архитектуры клиент–сервер. На сегодняшний день ECMAScript может предоставлять основные скриптовые возможности для множества сред выполнения, поэтому спецификация языка определяется без привязки к какой-либо конкретной среде.

Разберем две наиболее популярные среды выполнения.

Веб-браузер предоставляет среду выполнения ECMAScript для вычислений, производимых на стороне клиента, например, объекты, представляющие окна, меню, всплывающие окна, диалоговые окна, текстовые поля, привязки, фреймы, историю посещений страниц, cookies-файлы, а также ввод и вывод информации. Кроме того, среда выполнения предоставляет средства для подключения скриптового кода к таким событиям, как изменение фокуса, загрузка страницы и изображения, выход со страницы, ошибка и отмена, выбор, отправка формы, действия мышью. Скриптовый код появляется внутри HTML-кода, а отображаемая страница представляет собой, с одной стороны, сочетание элементов пользовательского интерфейса, а с другой стороны — текста и изображений, как статичных, так и вычисленных. Скриптовый код реагирует на действия пользователя, поэтому потребность в основной программе отсутствует.

Веб-сервер предоставляет другую среду выполнения для вычислений, производимых на стороне сервера, включая объекты, представляющие запросы, клиентов и файлы, а также механизмы для блокировки и совместного использования данных.

Благодаря использованию одновременно скриптов на стороне браузера и на стороне сервера можно распределить вычисления между клиентом и сервером и при этом обеспечить персонализированный пользовательский интерфейс для веб-приложения.

Вообще говоря, каждый веб-браузер и сервер, поддерживающий ECMAScript, предоставляет свою собственную среду выполнения, дополняя среду выполнения языка ECMAScript.

Среда выполнения определяет не только API и объекты среды, но и контекст безопасности. Скрипты, загруженные из Интернета и исполняемые в браузере, должны обладать весьма ограниченным набором прав, в частности, не должны иметь доступ к жесткому диску пользователя.

В приведенном выше описании попутно были введены понятия **API среды выполнения** и **объект среды выполнения**.

Бразуер — прикладное программное обеспечение для просмотра веб-страниц, содержания веб-документов, управления веб-приложениями, а также для решения других задач. В Глобальной сети браузеры используют для запроса, обработки, манипулирования и отображения содержания веб-сайтов.

В контексте данного изложения на браузер нужно смотреть как на платформу (среду выполнения) и, соответственно, необходимо разделить понятие «браузер», «браузерный движок» и «движок JavaScript». Как правило, браузер имеет **браузерный движок**, занимающийся отрисовкой страниц, и движок JavaScript; такая декомпозиция упрощает тестирование, переиспользование или использование в других проектах.

Браузерный движок (англ. *layout engine*) представляет собой программу, преобразующую содержимое веб-страниц (файлы HTML, XML, цифровые изображения и т. д.) и информацию о форматировании (в форматах CSS, XSL и т. д.) в интерактивное изображение форматированного содержимого на экране. Браузерный движок обычно используется в веб-браузерах (отсюда название), но может

использоваться и в почтовых клиентах или других программах, нуждающихся в отображении и редактировании содержимого веб-страниц.

Термин «браузерный движок» получил распространение после того, как движки стали отделимы от браузера. В число наиболее распространенных движков входят следующие: Gecko, KHTML, WebKit, Trident и Edge.

Таким образом, браузер имеет:

1) браузерный движок, который выполняет парсинг страниц, построение объектного дерева документа, раскладку элементов и визуализацию;

2) движок JavaScript, который выполняет JavaScript-код;

3) прочие составные части, которые, например, работают с сетью, выполняют шифрование и т. д.

Кроме того, браузер обеспечивает среду выполнения, предоставляя браузерные API и объекты среды.

Node.js — программная платформа, основанная на движке V8. Node.js, добавляет возможность JavaScript взаимодействовать с устройствами ввода–вывода через свой API (написанный на C++), подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода.

Скриптовый язык — это язык программирования, используемый для управления средствами существующей системы, а также для их настройки и автоматизации. В таких системах пользовательский интерфейс уже имеет набор полезных функциональных возможностей, а скриптовый язык является лишь механизмом, позволяющим ими программно управлять. Таким образом, считается, что существующая система обеспечивает среду объектов и средств, дополняющую возможности скриптового языка.

1.3. Общая характеристика языка

В документации указано, что ECMAScript представляет собой объектно-ориентированный язык программирования, предназначенный для проведения вычислений с объектами и управления этими

объектами в среде выполнения. В соответствии с официальным определением язык ECMAScript не предназначен для использования в качестве самодостаточной вычислительной системы — в спецификации даже не описаны средства для ввода внешних данных или вывода результатов вычислений; однако предполагается, что вычислительная среда для программы на языке ECMAScript предоставит не только объекты и иные средства, описанные в спецификации, но и некоторые объекты, определяемые конкретной средой.

Читатель может удивиться. Действительно, первое, что используется при изучении Node.js, — это средства ввода и вывода, console.log('некоторый текст').

Дело в том, что в самом ECMAScript объект console и его метод log не описаны, это объекты среды — Node.js. Именно среда предоставляет объекты для консольного ввода и вывода.

ECMAScript является объектным языком: базовый язык и средства выполнения представлены объектами, а программа на ECMAScript — это совокупность общающихся объектов. Объект в ECMAScript представляет собой набор свойств, каждое из которых обладает атрибутами в количестве ноль или более, которые определяют, как каждое из свойств может использоваться: например, когда атрибут Writable какого-либо свойства установлен в false, попытка выполняемого кода на ECMAScript изменить значение этого свойства не увенчается успехом.

Свойства являются контейнерами, в которых содержатся примитивные значения, другие объекты или функции. Примитивное значение — это элемент одного из следующих встроенных типов: Undefined, Null, Boolean, Number, String. Объект — это элемент встроенного типа Object. Функция — это вызываемый объект. Частным случаем функции является метод, т. е. функция, связанная с объектом через его свойство.

ECMAScript определяет набор встроенных объектов, завершающих определение сущностей ECMAScript. К этим встроенным объектам относятся объект global object, объект Object, объект Function, объекты Array, String, Boolean, Number, Math, Date, RegExp, JSON, а также объекты Error: Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError и URIError.

Язык ECMAScript не содержит классов, подобных классам в языках C++, Smalltalk или Java. Объекты могут создаваться различными способами, в том числе посредством буквенного обозначения или с помощью конструкторов, которые создают объекты и выполняют код, инициализирующий их полностью или частично путем присвоения их свойствам начальных значений.

Каждый конструктор является функцией, которая обладает свойством «prototype», используемым для реализации прототипного наследования и разделяемых свойств (подробнее о прототипах и конструкторах будет рассказано в следующей главе). Для создания объектов используются конструкторы в выражениях new, например: new Date (2009, 11) создает новый объект Date. Последствия вызова конструктора без использования new зависят от этого конструктора. Например, вызов Date() создает не объект, а строковое представление текущей даты и времени.

Каждый объект, созданный конструктором, содержит неявную ссылку (называемую прототипом объекта) на значение свойства «prototype» его конструктора. В свою очередь, прототип может обладать неявной ссылкой (имеющей значение, отличное от null) на свой прототип и т. д. Это называется цепочкой прототипов. Когда используется ссылка на свойство объекта, она является ссылкой на свойство с таким именем у первого объекта цепочки прототипов, который содержит свойство с таким именем. Иными словами, сначала на наличие этого свойства обследуется непосредственно указанный объект, и если этот объект содержит свойство с таким именем, то значит ссылка указывает на это свойство. В противном случае указанное свойство ищется в прототипе этого объекта и т. д.

В целом в объектно-ориентированных языках, использующих классы, состояние заключено в экземплярах, методы находятся в классах, а наследуется только структура и поведение. В ECMAScript и состояние, и методы находятся в объектах. Поэтому наследуются и структура, и поведение, и состояние.

Все объекты, прямо не содержащие какого-либо конкретного свойства, которое содержит их прототип, разделяют это свойство и его значение. Это проиллюстрировано на рис. 1.

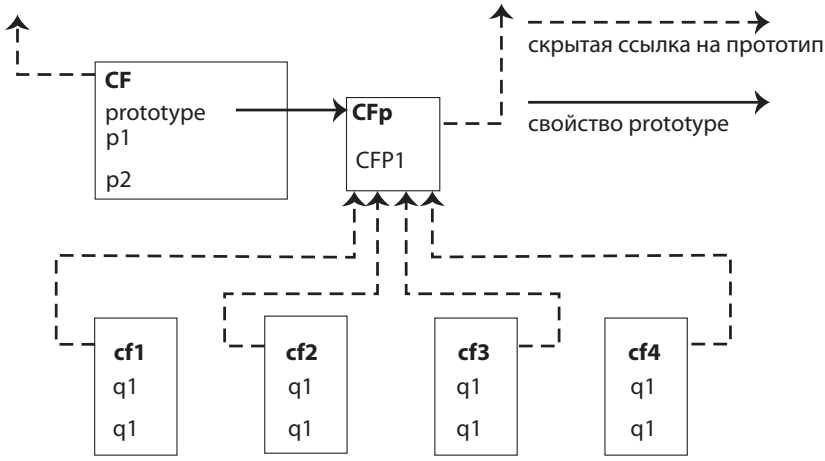

Рис. 1. Взаимосвязь между конструктором, прототипом и объектами

CF является конструктором, а также объектом. С использованием выражений new было создано пять объектов: cf1, cf2, cf3, cf4 и cf5, каждый из них содержит свойства с именами q1 и q2. Пунктирной линией обозначены отношения неявного прототипного наследования. Например, прототипом cf3 является CFp. Сам конструктор CF обладает двумя свойствами — P1 и P2, которые не видны для CFp, cf1, cf2, cf3, cf4 и cf5. Свойство CFP1 в CFp разделяют cf1, cf2, cf3, cf4 и cf5 (а CF не разделяет этого свойства), как и все свойства в неявной цепочке прототипов CFp с именами, отличными от q1, q2 или CFP1. Обратите внимание на отсутствие неявной прототипной связи между CF и CFp.

В отличие от объектно-ориентированных языков, использующих классы, в языке ECMAScript свойства могут добавляться к объектам динамически, путем присвоения им значений. То есть конструкторы не обязаны указывать имена или присваивать значения всем или некоторым свойствам создаваемого объекта. В схему, приведенную выше, можно добавить новое разделяемое свойство для cf1, cf2, cf3, cf4 и cf5, присвоив свойству CFp новое значение.

Контрольные вопросы

1. Начинающий JavaScript-разработчик, желая понять, как работает функция setTimeout, обратился к документации ECMA-262 [1], но не нашел там интересующей его информации. Почему? В какой документации можно узнать о функции setTimeout?

2. Среда выполнения определяет среди прочего объекты среды и контекст безопасности. Приведите примеры, которые показывают, что объекты, предоставляемые бразуером и Node.js, отличаются. Одинаковый ли контекст безопасности этих двух сред?

Глава 2
ПРОТОТИПНОЕ НАСЛЕДОВАНИЕ В JAVASCRIPT

Наследование в JavaScript сильно отличается от наследования в объектно-ориентированных языках, основанных на классах, таких как C++ и Java. В C++ и Java есть классы, являющиеся элементом программного кода, описывающего абстрактный тип данных и его частичную или полную реализацию. В JavaScript понятие класса отсутствует.

2.1. Прототипное наследование встроенных объектов

Фундаментальную концепцию прототипного наследования в JavaScript будем разбирать на примерах. Создадим, например, два массива:

```
let arr1 = new Array();
let arr2 = new Array(1, 2, 3, 4);
```

Здесь Array — это функция, которая используется как конструктор (т. е. для создания объекта), arr1 и arr2 — объекты. По сути, функция Array мало чем отличается от созданной ниже функции User:

```
function User(name, age){
    this.name = name;
    this.age = age;
};
admin = new User('Vova', 30);
accountant = new User('Klara', 50);
```

Отличие в том, что Array — встроенный объект (напомним, что функция — это вызываемый объект, англ. *callable object*), а User — создана нами. Использование же функций Array и User как конструкторов в приведенных выше фрагментах кода вообще ничем не отличается.

Есть договоренность, что название функции-конструктора пишут с прописной буквы, а объектов, созданных с помощью конструкторов, — со строчной.

Кроме того что функцию можно использовать как конструктор, можно работать с функцией и как с невызывабельным объектом (т. е. без учета ее функциональной специфики). Например, можно задать свойство функции:

```
User.prop = 'Static property';
```

Это свойство будет «принадлежать» самой функции, а не созданным с ее помощью объектам admin и accountant.

Итак, мы создали объекты arr1, arr2, …, arr10. Откуда у них методы типа join, pop, push, shift, unshift и другие массивные методы? Зачастую примитивные учебники по ООП учат, что «объекты содержат свойства и методы для работы с этими свойствами». Может сложиться ложное убеждение, что методы хранятся в объектах arr1, arr2, …, arr10. Если бы методы хранились в объектах arr1, arr2, …, arr10 (т. е. в оперативной памяти, выделенной для объектов, хранился бы код соответствующих функций join, pop, push), то это было бы неэффективно с точки зрения использования памяти. Действительно, если бы мы создали несколько объектов типа массив, то столько же раз в памяти надо было бы размещать код методов (ведь функция — это какой-то код и его надо хранить).

Методы хранятся в прототипе, который можно представить в виде «мешка» с полезными методами.

На рис. 2 изображены три сущности:

1) Array: Function — функция-конструктор;

2) arr1, arr2: Array — экземпляры;

3) Array.prototype: Object.

Установим связи между этими сущностями. Выше мы задавали свойство prop функции User, аналогично можно определить свойство и у встроенной функций Array. Оказывается, что у Array изначально есть свойство prototype. Значением свойства prototype является ссылка на Array.prototype — объект, являющийся прототипом всех массивов (см. рис 2, стрелка, которая идет от Array к Array.prototype).

В свою очередь, у прототипа Array.prototype есть свойство constructor (см. рис 2, стрелка, которая идет от Array.prototype к Array). Значением этого свойства является ссылка на функцию Array — ту, которая является конструктором всех объектов типа Array. Сказанное не означает, что typeof arr1 == Array.

У объекта arr1 (как и у всех объектов) есть внутреннее свойство [[Prototype]], значением которого является ссылка на

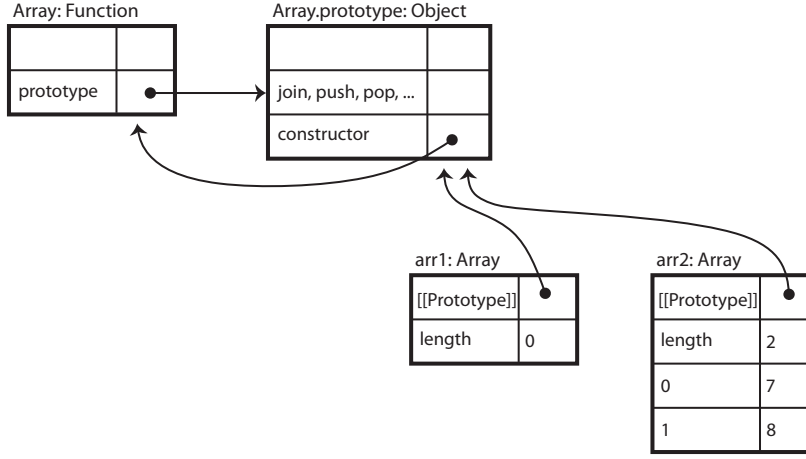

Рис. 2. Прототипное наследование встроенных объектов

Array.prototype. (Не путать внутреннее свойство [[Prototype]] и свойство prototype!)

Единственным (точнее, почти единственным) предназначением свойства prototype является установление прототипа у вновь создаваемых объектов, т. е. присвоение значения внутреннему свойству [[Prototype]].

По сути, значение свойства prototype (Array.prototype) копируется в arr1.[[Prototype]] в момент создания объекта arr1 = new Array() (на самом деле именно так оно и работает «под капотом»).

Введем понятие «собственных» свойств. У объекта arr1 по числовым индексам 0, 1, … хранятся значения, есть свое свойство length — это собственные свойства. Надо полагать, что есть и несобственные свойства? Да, если объект arr1 не имеет некоторого свойства (или метода[1]), а к этому свойству обращаются в программном коде, то движок JavaScript ищет соответствующее свойство в прототипе объекта. То есть движок переходит от объекта arr1 к объекту arr1.[[Prototype]]. Что будет, если и в arr1.[[Prototype]] соответствующего свойства нет, разберем далее.

Методы join, push, pop и т. д. являются свойствами Array.prototype; они не являются «собственными» свойствами объектов arr1, arr2, …, arr10. Когда в коде происходит обращение к методу arr1.join(), соответствующий метод находится в прототипе и вызывается в контексте объекта arr1. Можно сказать, что методы join, push, pop и т. д. разделяются между объектами arr1, arr2, …, arr10, тем самым экономится память.

Проведем эксперимент. Создадим дополнительные свойства и методы у объекта arr1.

```
arr1.myproperty = 'abc';
console.log(arr1.myproperty);
console.log(arr2.myproperty); //underfined
```

Здесь myproperty — это «собственное» свойство arr1; оно не является членом прототипа.

[1] Метод — это, по сути, тоже свойство, которое является функцией.

Продолжим изучение связей между встроенными объектами и расширим диаграмму прототипного наследования (см. рис. 2). Создадим объект obj1:

```
obj1 = new Object();
```

Все, что ранее было написано про функцию-конструктор Array, прототип массивов Array.prototype и экземпляры массивов arr1 и arr2, можно по аналогии перенести на функцию-конструктор Object, прототип объектов Object.prototype и экземпляры объектов obj1 и obj2.

Если в коде написать arr1.hasOwnProperty(), то такой метод найдется. Возникает вопрос, где же этот метод находится? Очевидно, что в самом объекте его (т. е. метода hasOwnProperty) нет, но и в Array.prototype его тоже нет. Array.prototype является объектом, и у него тоже (как у arr1) есть внутреннее свойство [[Prototype]], значением которого является Object.prototype (см. рис. 3, прямоугольник Array.prototype.[[Prototype]], стрелка из него ведет к Object.prototype).

Проводя аналогию с иерархией классов (наследованием классов), говорим, что Array.prototype прототипно наследует от Object.prototype; arr1 прототипно наследует от Array.prototype. Теперь можно дать ответ на поставленный вопрос: метод hasOwnProperty найдется в прототипе Array.prototype, которым является Object.prototype.

У Object.prototype есть свойство [[Prototype]], значением которого является null.

Array и Object — это не только функции, но еще и объекты, значит, у них есть свойство [[Prototype]]. Выясним, на что указывает (чему равно) свойство [[Prototype]] объектов-функций Array и Object.

Вспомним, что наряду с декларацией функций типа

```
function myFun(foo, bar){
    /*some code*/
}
```

можно создавать функции путем вызова конструктора:

```
myFun = new Function();
```

Второй способ, по сути, ничем не отличается от ранее разобранного: arr1 = new Array(), obj1 = new Object().

Функции — это объекты, и у них есть методы, например, call, apply, bind. Так, можно писать

```
myFun.call(someObj, arg1, arg2);
```

Откуда эти методы берутся? Где они хранятся? Названные методы являются членами прототипа — Function.prototype (рис. 4: Function, Function.prototype).

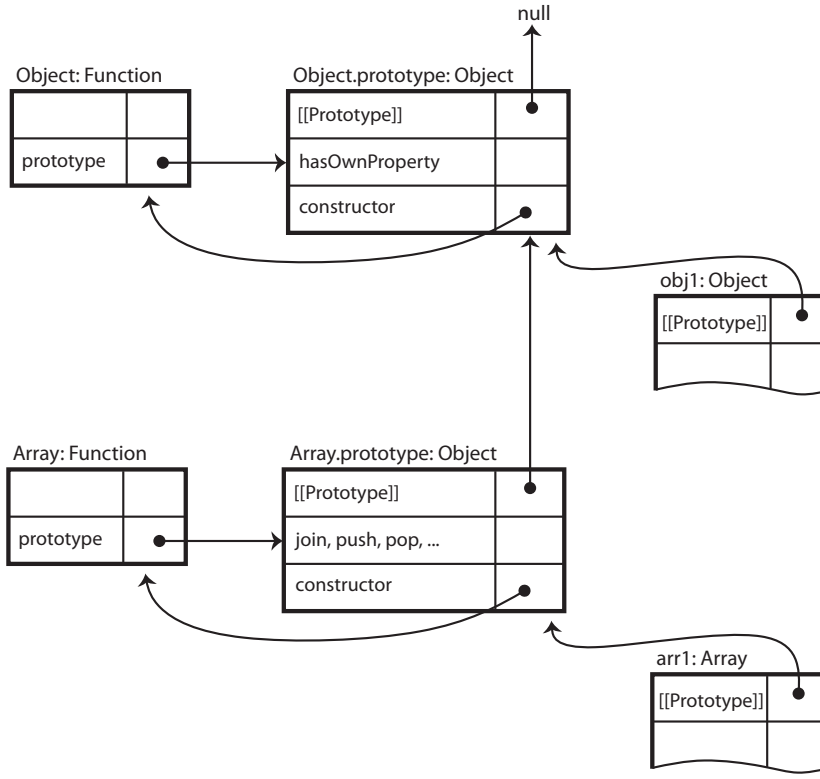

Рис. 3. Прототипное наследование встроенных объектов

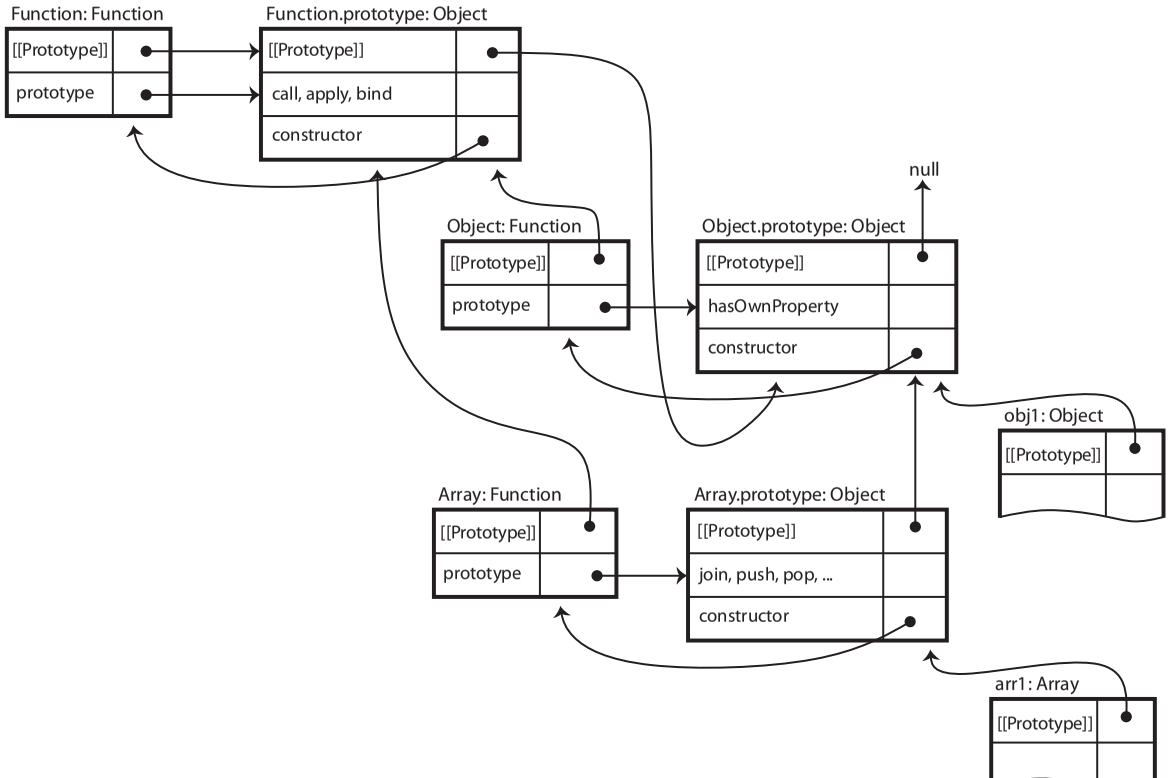

Рис. 4. Прототипное наследование встроенных объектов

Так же как arr1 прототипно наследовал от Array.prototype, obj1 — от Object.prototype, так и описанные нами функции User, myFun и встроенные Object, Array прототипно наследуют от Fuction.prototype (рис. 4: стрелки от Object.[[Prototype]] и Array. [[Prototype]] к Function.prototype).

Зададимся вопросом: что является значением внутреннего свойства Function.prototype.[[Prototype]]? Function.prototype является объектом, следовательно, Function.prototype прототипно наследует от Object.prototype, точно так же как Array.prototype прототипно наследуют от Object.prototype.

Function — это функция, такая же, как Object, Array; значением свойства [[Prototype]] является ссылка на Function.prototype (см. рис. 4).

Следует отметить, что эта диаграмма не может быть расположена на плоскости так, чтобы наследники (прототипные) оказались ниже своих предков.

2.2. Операторы instanceof и typeof

Во многих тестах по JavaScript встречаются вопросы про операторы instanceof и typeof, на собеседованиях при приеме на должность фронтенд-разработчиков такие вопросы задают часто. Приведем примеры:

```
str1 = 'abcd';
str2 = new String();
str3 = new String('efgh');
console.log(str1 instanceof String);
console.log(str2 instanceof String);
console.log(str3 instanceof String);
console.log(str3 instanceof Object);
console.log(Function instanceof Object);
console.log(Object instanceof Function);
```

Нередко в справочниках и учебниках приводятся неверные определения, которые запутывают читателей. Приведем два примера:

— оператор instanceof проверяет, создан ли данный объект с помощью данного конструктора;

— операция instanceof возвращает true, если объект относится к данному классу объектов.

Используя схемы на рис. 4, понять принцип работы оператора instanceof не составит труда!

Пусть необходимо проверит истинность выражения:

Объект instanceof Конструктор

Заметим, что если слева от instanceof не объект, а примитив, или справа не вызываемый объект (т. е. не функция), результатом проверки будет ошибка.

Берем объект, смотрим в его прототип, там ищем свойство constructor. Если свойство constructor нашлось, то сравниваем найденное значение и правый операнд; если они совпали, то результат выполнения оператора instanceof равен true. Если не совпали или свойство constructor не нашлось в прототипе, то переходим к вышестоящему прототипу и повторяем описанные шаги. В итоге есть два варианта:

— либо мы найдем тот прототип, значение свойства constructor которого совпадает с правым операндом, и тогда результат выполнения оператора instanceof равен true;

— либо, двигаясь по цепочке прототипов, дойдем до null, и тогда результат выполнения оператора instanceof равен false.

Разберем вышеприведенные примеры из тестов. Переменная str2 была создана с помощью конструктора String, поэтому str2 instanceof String равен true.

Переменная str3 также была создана с помощью конструктора String, отличие лишь в том, что ее примитивное значение равно 'efgh', а у str2 — пустая строка. В результате str3 instanceof String равен true.

Проверяем console.log(str3 instanceof Object); //true:

1. Проверяем str3.[[Prototype]], это String.prototype — объект, у которого есть свойство constructor со значением String, который не равен правому значению оператора, т. е. Object, значит, надо двигаться дальше по цепочке прототипов. Переходим от String.

prototype к его прототипу. String.prototype — это объект, его прототипом является объект Object.prototype, формально String.prototype.[[Prototype]] == Object.prototype.

2. У Object.prototype есть свойство constructor со значением Object. На этом проверка заканчивается, результат — true.

Проверяем console.log(str3 instanceof Array); //false:

1. Проверяем str3.[[Prototype]], это String.prototype — объект, у которого есть свойство constructor со значением String, который не равен правому значению оператора, т. е. Array, значит, надо двигаться дальше по цепочке прототипов. Переходим от String.prototype к его прототипу. String.prototype — это объект, его прототипом является объект Object.prototype, формально String.prototype.[[Prototype]] == Object.prototype.

2. У Object.prototype есть свойство constructor со значением Object, который не равен правому значению оператора, т. е. Array, значит, надо двигаться дальше по цепочке прототипов. Переходим от Object.prototype к его прототипу. Object.prototype — это объект, его прототипом является null, формально Object.prototype.[[Prototype]] == null.

3. Проверка заканчивается, результат — false.

Проверяем:

```
str1 = 'abcd';
console.log(str1 instanceof String);
```

Правда ли, что str1 — это объект, у которого есть свойство [[Prototype]]? Нет, неправда, str1 — это примитивное значение, поэтому str1 instanceof String вернет false.

Возникает вопрос: если str1 — это примитивное значение, то почему можем написать str1.indexOf('bc')? Откуда у примитивного значения есть метод? Дело в том, что движок JavaScript, встречая такую запись, на лету «под капотом» создает временный объект (строку с примитивным значением 'abcd'), у него вызывает метод indexOf, после чего временный объект удаляется сборщиком мусора.

2.3. Свойство __proto__ и метод getPrototypeOf()

Значение внутреннего свойства [[Prototype]] можно получить в программном коде. Обратиться к нему можно с помощью свойства __proto__. Например:

```
myArr = new Array();
console.log(myArr.__proto__);
console.log(myArr.__proto__ == Aray.prototype);
console.log(Array == Aray.prototype.constructor);
console.log(myArr.__proto__.__proto__ ==
   Object. prototype);
```

Прототип также можно получить с помощью метода getPrototypeOf().

Метод Object.getPrototypeOf() возвращает прототип (т. е. внутреннее свойство [[Prototype]]) указанного объекта. Приведем примеры.

```
var proto = {};
var obj = Object.create(proto);

Object.getPrototypeOf(obj) === proto;
// Это true
> Object.getPrototypeOf('foo') // код ES5
TypeError: "foo" is not an object
> Object.getPrototypeOf('foo') // код ES6
String.prototype
```

В ECMAScript5, если параметр obj не является объектом, программа выдает исключение TypeError. В ECMAScript6 параметр будет приведен к объекту Object.

2.4. Прототипное наследование пользовательских объектов

С использованием прототипов можно организовать иерархию, подобную иерархии классов в C++ или Java.

```
Let Point = function(x, y){
    this.x = x;
    this.y = y;
};
Point.prototype.move = function(deltaX, deltaY){
    this.x += deltaX;
    this.y += deltaY;
};
// Наследование
let ColorPoint = function(x, y, color){
    Point.call(this, x, y);
    this.color= color;
}
ColorPoint.prototype =
Object.create(Point.prototype);
ColorPoint.prototype.constructor =
ColorPoint;

// Инстанцирование
let myPoint = new Point(1, –1);
let myColorPoint = new ColorPoint(4, 3, 'red');
myColorPoint.move(2,4);
console.log(myColorPoint.x, myColorPoint.y) // 6, 7
```

Вместо строк

```
ColorPoint.prototype =
Object.create(Point.prototype);
ColorPoint.prototype.constructor =
ColorPoint;
```

можно написать

```
ColorPoint.prototype.__proto__ = Point;
```

Контрольные вопросы

1. В исходных кодах мы иногда пишем

```
String.fromCharCode(код).
```

Чей это метод? Является ли метод fromCharCode членом прототипа? Как его изобразить на схеме, по аналогии с тем, как были изображены методы join или call (см. рис. 4)?

2. Есть встроенный объект Math. Мы не создаем экземпляры, объекты типа Math. Пишем Math.exp(1). А как Math изображается на диаграмме (см. рис. 4)?

3. В языке Python можно умножить строку на число, строка повторится соответствующее число раз (конкатенируется сама с собой). Такого метода в JavaScipt нет. Написать строковый метод repeat, который принимает число на вход, чтобы работало так:

```
let s1 = 'abc';
console.log(s1.repeat(2)),
```

вывод: abcabc.

4. Что выведет на экран данная программа:

```
myStr = 'hello';
console.log(myStr instanceof String);
myObj = {};
console.log(myObj instanceof Object);
myArr = [1, 2, 3];
console.log(myArr instanceof Object);
```

Ответы

1. Метод fromCharCode — статический метод String, он не является членом прототипа.

4. false, true, true.

Строка myStr — это примитивное значение, а не объект, имеющий свойство [[Prototype]]. Запись myObj = {} — это краткая форма записи myObj = new Object({}), т. е. хотя запись внешне похожа на myStr = 'hello', в данном случае «под капотом» используется конструктор Object.

Глава 3
SAME ORIGIN POLICY. ЭКСПЕРИМЕНТЫ С КРОССДОМЕННЫМ ВЗАИМОДЕЙСТВИЕМ

Same Origin Policy — важная концепция безопасности в вебе. Перед тем как изучать ее, проведем несколько экспериментов.

3.1. Эксперименты с Same Origin Policy

Чтение и запись Document Object Model

Цель атаки — получить конфиденциальные данные пользователя, авторизированного на сайте bank.com. Атакующий создает в сети свой сайт attacker.com с контентом, привлекательным для жертвы. На сайте attacker.com расположен iframe, в который грузится bank.com; размер iframe может быть 1 × 1 пиксель, а потому пользователь его даже не заметит и ни о чем не заподозрит.

```
<html>
Тут привлекательный контент…
<iframe src="https://bank.con" id="victimlframe"> </iframe>
</html>
```

Пререквизитом атаки является то, что пользователь-жертва авторизовался на сайте bank.com, скажем, в первой вкладке браузера. Далее атакующий заманивает пользователя-жертву на свой сайт

attacker.com (например, присылает ссылку в письме), и пользователь открывает сайт attacker.com во второй вкладке браузера.

Когда браузер парсит страницу attacker.com, он встречает тег iframe и, проанализировав содержимое атрибута src, делает соответствующей запрос на bank.com. Браузер автоматически добавляет к этому запросу авторизационные cookie. Отметим, что cookie добавляются по адресу назначения запроса, а не источника! То есть браузер добавляет cookie, соответствующие bank.com, а не attacker.com.

Как только содержимое загрузилось в iframe, скрипт пытается считать содержимое DOM из iframe:

```
<iframe src="https://www.wikipedia.org" id="victimIframe"></iframe>
<script>
$(document).ready(function(){
    var iframeContent = document.getElementById("victimIframe").
contentDocument;
    console.log(iframeContent);
});
</script>
```

Исходя из того, что скрипт может читать содержимое тега div, можно предположить, что у скрипта есть доступ к содержимому iframe:

```
var divContent = docment.getElementById("myDiv").innerHTML;
console.log(divContent);
```

На самом деле данный код вызовет ошибку:

> Uncaught DOMException: Failed to read the 'contentDocument' property from 'HTMLIFrameElement': Blocked a frame with origin "http://attacker.com" from accessing a cross-origin frame.

Причина в том, что браузер не позволяет производить чтение и запись DOM источнику, отличному от источника iframe (окна).

Чтение ответов Ajax-запросов через XMLHttpRequest и fetch

Цель атаки — получить конфиденциальные данные пользователя, авторизированного на сайте facebook.com. Атакующий создает в сети свой сайт attacker.com и заманивает туда жертву. На сайте attacker.com в теге <script> расположен JavaScript-код, который пытается украсть личную переписку с Facebook, делая следующий Ajax-запрос на чтение личных сообщений:

```
$.ajax({
    url: "https://facebook.com/loadMessages",
    type: "GET",
    xhrFields: { withCredentials: true }
}, function(data) {
    console.log(data);
});
```

Пререквизитом атаки является то, что пользователь-жертва авторизовался на сайте bank.com, скажем, в первой вкладке браузера, а во второй вкладке открыл сайт attacker.com. Напомним, что браузер, делая Ajax-запрос, автоматически добавляет к этому запросу авторизационные cookie.

Попытка выполнить данный код во вкладке attacker.con приведет в ошибке:

> XMLHttpRequest cannot load https://facebook.com/. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'https://attacker.com' is therefore not allowed access

Это Same Origin Policy в действии: запрос был успешно отправлен на Facebook с cookie пользователя, и Facebook отправил в ответ личные сообщения. Однако браузер не отдает JavaScrip-коду ответ сервера.

Как именно реализуется Same Origin Policy при осуществлении кроссдоменных Ajax-запросов, мы разберем в следующих параграфах, когда будем обсуждать CORS.

3.2. Понятие Same Origin Policy

Same Origin Policy (в переводе с англ. «принцип одинакового источника») — политика, определяющая, как документ или скрипт, загруженный из одного источника (origin), может взаимодействовать с ресурсом из другого источника (origin). Это важная концепция безопасности, позволяющая изолировать потенциально зловредные документы.

Same Origin Policy разрешает клиентским сценариям, находящимся на страницах одного сайта, доступ к методам и свойствам друг друга без ограничений, но предотвращает доступ к большинству методов и свойств для страниц на разных сайтах.

Same Origin policy ограничивает доступ JavaScript-коду, загруженному из одного источника:

— к чтению ответов Ajax-запросов через XMLHttpRequest и fetch из другого источника;

— чтению и записи Document Object Model (DOM) другого источника;

— чтению и записи хранимых данных (cookie, session & local storage) другого источника.

Две страницы имеют одинаковый origin (источник), если протокол, порт (если указан) и хост одинаковы для обеих страниц. В табл. 1 даны примеры origin-сравнений с URL http://company.com/dir/page.html.

Появление концепции Same Origin Policy относится к Netscape Navigator 2, 1995 г. Изначально политика была разработана для предотвращения доступа к объектной модели документа, но впоследствии была расширена для того, чтобы защитить критические части глобального объекта JavaScript.

Таблица 1

Сравнение URL-адресов в Same Origin Policy

URL	Совпадение	Причина отличий
http://company.com/dir2/other.html	Да	
http://company.com/dir/another.html	Да	

Окончание табл. 1

URL	Совпадение	Причина отличий
https://company.com/secure.html	Нет	Разные протоколы
http://company.com:81/dir/etc.html	Нет	Разные порты
http://new.company.com/dir/ya.html	Нет	Разные хосты

З а м е ч а н и е. У Internet Explorer есть два исключения, касающихся Same Origin Policy.

Во-первых, в IE введено понятие «зона доверия», всего выделяется пять зон: Internet, Local Intranet, Trusted Sites, Restricted Sites, My Computer. Пользователи могут использовать эти зоны, чтобы легко обеспечить соответствующий уровень безопасности для различных типов веб-контента. Например, пользователь может полностью доверять сайтам в интрасети своего учреждения и, соответственно, снизить уровень требований к безопасности. Этот же пользователь может не доверять сайтам в Интернете, а потому назначить более высокий уровень требований к безопасности во всей зоне Интернета. Этот более высокий уровень безопасности не позволяет запускать активный контент и загружать код на свои компьютеры.

Если есть определенные веб-сайты, которым пользователь доверяет, он может (в настройках браузера) разместить отдельные URL-адреса или целые домены в зоне надежных сайтов.

Если оба домена находятся в зоне высокого доверия, то Same Origin Policy не применяется.

Во-вторых, IE не включает порт в компоненты Same Origin, http://company.com:81/index.html и http://company.com/index.html относятся к одному источнику, а потому никакие ограничения не применяются.

Эти исключения нестандартны и не поддерживаются ни в одном другом браузере, но их полезно знать при разработке веб-приложений на базе Windows RT или IE.

3.3. Кроссдоменные запросы не из скрипта

Следует отметить, что кроссдоменные запросы — это то, что составляет суть веба. Веб создавался для того, чтобы обеспечить пользователям легкую навигацию между сайтами, например, в результате щелчка по гиперссылкам.

1. Пусть пользователь просматривает в браузере документ, загруженный с сайта bank.com, и принимает решение перейти на сайт http://other.net, для чего в адресной строке во вкладке браузера вводит нужный адрес и нажимает «Enter». Далее:

а) браузер инициализирует GET http-запрос, origin которого отличается от текущего;

б) соответствующий контент принимается браузером от сервера other.net;

в) DOM, ассоциированный с bank.com, уничтожается, а текущий origin меняется на <http, other.net, 80>.

Разумеется, политики безопасности не должны и не могут мешать пользователю самому инициировать переход куда-либо путем ввода адреса в адресную строку браузера.

2. То же самое относится к переходу к иному ресурсу (cross-origin запросу) с помощью клика по гиперссылке. Пользователь вправе самостоятельно решать, куда ему переходить. SOP здесь не применяется.

3. Теперь рассмотрим кроссдоменные запросы, инициализированные браузером и, в известном смысле, прозрачные для пользователя. Например, браузер парсил страницу, загруженную с bank.com, и обнаружил тег для вставки изображения:

```
<img src='http://ico.bank.com' alt='logo'>
```

или

```
<img src='http://ico.other.net' alt='logo'>
```

Данный кроссдоменный запрос будет сделан, при этом будут выставлены следующие заголовки:

```
host: 'icons.bank.com'
connection: 'keep-alive'
pragma: 'no-cache'
'cache-control': 'no-cache'
'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98
Safari/537.36'
accept: 'image/webp, image/*,*/*; q=0.8'
referer: 'http://bank.com/'
accept-encoding: 'gzip, deflate, br'
accept-language: 'ru-RU, ru; q=0.9, en-US; q=0.8, en; q=0.7'
```

Возможно, что в другом браузере будут выставлены немного иные заголовки, но суть в том, что, запрашивая ресурсы с других доменов, браузер не выставлял заголовков, как-либо ограничивающих кроссдоменное взаимодействие.

4. Аналогично работают кроссдоменные запросы, инициализированные браузером к стилям CSS. Например, браузер парсил страницу, загруженную с bank.com, и обнаружил тег для загрузки стилей:

```
<link href='http://solod.webhost.com/style.css' type='text/css'
rel='stylesheet'>
```

Same Origin Policy не запрещает подключать из других источников JavaScript-код[2] через <script src="http://another-origin.com"></script>. Источник подключенного JavaScript-кода будет такой же, как источник страницы, которая его подключила. Это означает, что любая библиотека, подключаемая через тег <script>, получает доступ к ресурсам текущего источника, включая содержимое страницы и cookie.

Содержимое тегов <video>, <audio>, <object>, <embed>, <applet> также может быть из другого источника.

[2] Это открывает возможность для использования JSONP (JSON with padding) — дополнения к базовому формату JSON, предоставляющего способ запросить данные с сервера, находящегося в другом домене.

5. По прочтении некоторых недостаточно проработанных учебников может сложиться впечатление, что единственное предназначение SOP — ограничить потенциально опасные кроссдоменные запросы[3], инициализированные в JavaScript-коде. Это неверно.

Как мы видели в примере выше, SOP позволяет блокировать доступ к DOM дочерних iframe, содержащих контент с другого origin. Никаких Ajax-запросов там вообще нет.

Может показаться, что в примерах 3 и 4 браузер, не опасаясь, делал кроссдоменные запросы, так как адреса (значения атрибутов src и href), куда делать запросы, были получены из доверенного источника. Раз веб-разработчик вставил в главную страницу ссылки на внешние ресурсы, значит, он им доверяет.

Тут возникает вопрос: если кроссдоменная загрузка изображения инициирована в результате того, что скрипт сменил значение атрибута src:

```
function changeImg(){
    document.getElementById('myPicture').src =
    'https://otherdomain.net/foto.png';
}
```

будет ли браузер выставлять какие-либо специальные, связанные с безопасностью заголовки и как-то особым образом обрабатывать ответ сервера? Ответ: поведение браузера такое же, как в примере 3.

3.4. Кроссдоменные запросы: Ajax-запросы и CORS

При разработке веб-приложений, в частности с использованием фреймворка Angular, часто создают одностраничные приложения (Single Page Application) со следующей архитектурой:

— основной файл (обычно называемый index.html) и скрипты (ответственные, например, за манипуляции с DOM и осуществление

[3] Иногда более точно пишут: «потенциально опасные кроссдоменные Ajax-запросы».

Ajax-запросов) грузятся с веб-сервера, на котором запущен Angular. Этот фронтенд-сервер является точкой входа в приложение;

— запросы, связанные с осуществлением логики приложения (например, бронирование номеров в отеле), отправляются на другой веб-сервер — так называемый бэкенд-сервер. Эти Ajax-запросы отправляются, очевидно, на другой origin и потому попадают под определение кроссдоменных.

Политика SOP явно запрещает осуществление подобных кроссдоменных запросов из JavaScript для повышения безопасности веб-приложений. Однако в данном случае эти запросы необходимы. Для ослабления жестких требований SOP была предложена спецификация CORS.

Спецификация CORS (Cross-Origin Resource Sharing) была предложена в 2009 г., хотя уже в 2005 г. Консорциумом Всемирной паутины (W3C) были опубликованы работы на эту тему (см.: Authorizing Read Access to XML Content Using the <?access-control?> Processing Instruction 1.0). Спецификация 2009 г. признана устаревшей (англ. *obsolete*) и претерпела неоднократные обновления. Технология CORS является стандартом W3C [2].

CORS — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность браузеру получать разрешения на доступ к выбранным ресурсам с сервера на источнике (origin), отличном от того, что используется в данный момент.

В целях безопасности браузеры ограничивают Cross-Origin-запросы, инициируемые скриптами. Например, XMLHttpRequest и Fetch API следуют политике одного источника, а это значит, что веб-приложения, использующие такие API, могут запрашивать HTTP-ресурсы только с того домена, с которого были загружены, пока не будут использованы CORS-заголовки.

Оказывается, что не только Ajax-запросы используют CORS. Именно CORS применяется для разрешения кроссдоменных HTTP-запросов, необходимых для загрузки:

— шрифтов Web Fonts (для кроссдоменного использования шрифтов в @font-face в рамках CSS), чтобы серверы могли разворачивать TrueType шрифты, которые могут быть загружены только в кросссайтовой манере и использованы веб-сайтами, которым это разрешено;

— WebGL-текстур;

— фреймов с изображениями/видео, добавленными в канвас с помощью drawImage;

— стилей (для CSSOM доступа);

— скриптов (для отключенных исключений).

Кроме того, CORS применяется для вызова Fetch APIs или XMLHttpRequest в кросссайтовой манере, как описано выше.

Стандарт Cross-Origin Resource Sharing работает с помощью добавления новых HTTP-заголовков, позволяющих серверам описывать набор источников, которым разрешено читать информацию, запрашиваемую браузером. В частности, для методов HTTP-запросов, которые могут привести к побочным эффектам над данными сервера (в частности, для HTTP-методов, отличных от GET, или для POST-запросов, использующих определенные MIME-типы), спецификация требует, чтобы браузеры отправляли предварительный запрос (англ. *preflight*), запрашивая с помощью HTTP-метода OPTIONS у сервера список поддерживаемых методов, а затем, в случае подтверждения от сервера, отсылали фактический запрос с фактическим методом HTTP-запроса. Серверы также могут оповещать клиентов, должны ли «полномочия» (англ. *credentials*), включая Cookies и HTTP Authentication данные, быть отправлены с запросом.

Далее рассмотрим три возможных сценария обмена данными между браузером и сервером, а также проанализируем использование соответствующих HTTP-заголовков.

Простые запросы

Некоторые запросы не заставляют срабатывать CORS preflight. Они обычно называются «простыми запросами» [3], хотя Fetch-спецификация, определяющая CORS, не использует этот термин. Запрос, для которого не срабатывает CORS preflight (так называемый «простой запрос») — это запрос, удовлетворяющий следующим условиям:

1. Допустимые методы для запроса:

а) GET,

б) HEAD,

в) POST.

2. Кроме заголовков, которые автоматические проставляются браузером (например, Connection, User-Agent или любой другой заголовок с именем, определенным в спецификации метода Fetch в секции «Запрещенные имена заголовков (которые нельзя изменить программно)»), допустимыми заголовками, которые могут быть проставлены вручную, являются те заголовки, которые определены спецификацией метода Fetch как «CORS-безопасные заголовки запроса», такие как:

а) Accept,
б) Accept-Language,
в) Content-Language,
г) Content-Type (но учитывайте пункт 3),
д) Last-Event-ID,
е) DPR,
ж) Save-Data,
з) Viewport-Width,
и) Width.

3. Допустимыми значениями заголовка Content-Type являются:

а) application/x-www-form-urlencoded,
б) multipart/form-data,
в) text/plain.

4. Не должны быть зарегистрированы обработчики событий на любой объект XMLHttpRequestUpload, используемый в запросе; это достигается использованием свойства XMLHttpRequest.upload.

5. В запросе не должен использоваться объект типа ReadableStream.

Это те же типы межсайтовых запросов, которые браузеры могли выдавать еще до появления XMLHttpRequest-объектов (см. параграф 3.3), а потому старые веб-серверы, старающиеся предотвратить подделку межсайтовых запросов (англ. *Cross-Site Request Forgery*), не столкнутся ни с чем новым в плане контроля доступа HTTP.

Примечание. В WebKit Nightly и Safari Technology Preview добавлены дополнительные ограничения на значения, допустимые в заголовках Accept, Accept-Language и Content-Language. Если какой-либо из этих заголовков имеет «нестандартные» значения, WebKit/Safari не считает, что запрос удовлетворяет условиям «про-

стого запроса». Что именно WebKit / Safari считает «нестандартными» значениями для этих заголовков, официально не документировано. Ни один другой браузер не реализует эти дополнительные ограничения, потому что они не являются частью спецификации.

П р и м е р. Содержимое домена http://foo.example хочет обратиться к содержимому http://bar.other. На домене foo.example может использоваться следующий JavaScript-код:

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/public-data/';

function callOtherDomain() {
    if(invocation) {
        invocation.open('GET', url, true);
        invocation.onreadystatechange =
handler;
        invocation.send();
    }
}
```

Это приведет к простому обмену запросами между клиентом и сервером, с использованием CORS-заголовков для обработки привилегий (рис. 5).

Браузер отправит в таком случае на сервер следующий запрос:

```
GET /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U;
Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130
Minefield/3.1b3pre
Accept: text/html, application/xhtml+xml,
application/xml; q=0.9,*/*; q=0.8
Accept-Language: en-us, en; q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859–1,*; q=0.7
```

```
Connection: keep-alive
Referer: http://foo.example/examples/
access-control/simpleXSInvocation.html
Origin: http://foo.example
```

Сервер пришлет такой ответ:

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml

[XML Data]
```

Самым важным для нас заголовком здесь является Origin. Данный заголовок указывает, что запрос пришел с домена http://foo.example.

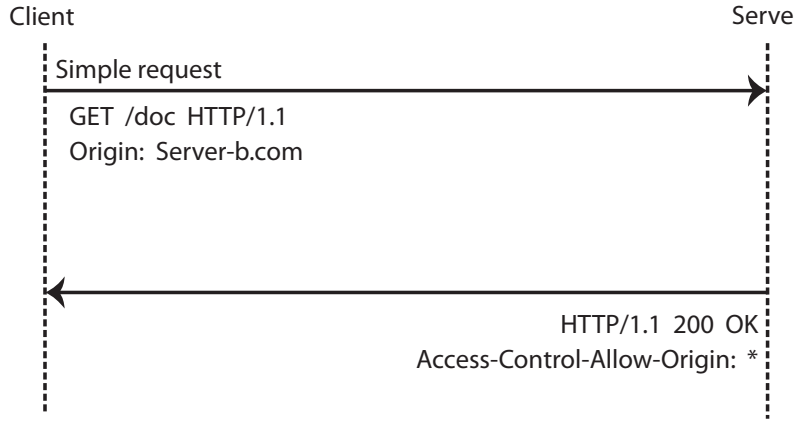

Рис. 5. Клиентские и серверные http-заголовки, реализующие CORS. Простой запрос

В ответ сервер отправляет обратно заголовок Access-Control-Allow-Origin. Использование заголовков Origin и Access-Control-Allow-Origin показывает протокол управления доступом в его простейшем варианте. В данном случае сервер отвечает: Access-Control-Allow-Origin: *, это означает, что к ресурсу может обращаться любой домен в межсайтовом режиме. Если владельцы ресурсов по адресу http: //bar.other хотят ограничить доступ к источнику только запросами с http: //foo.example, то они отправят обратно:

```
Access-Control-Allow-Origin: http://foo.example
```

Обратите внимание, что теперь ни один домен, кроме http: //foo.example (идентифицируемый заголовком Origin), не может получить доступ к ресурсу межсайтовым способом.

Если сервер хочет разрешить доступ более чем одному домену, то в ответе сервера должно быть по одной строчке Access-Control-Allow-Origin для каждого домена.

```
Access-Control-Allow-Origin: http://www.a.com
Access-Control-Allow-Origin: http://www.b.com
Access-Control-Allow-Origin: http://www.c.com
```

На практике чаще используется запись из нескольких доменов, разделенных пробелом:

```
Access-Control-Allow-Origin: http://www.a.com http://www.b.com
http://www.c.com
```

Запросы, требующие предварительной проверки

В кроссдоменном Ajax-запросе можно использовать не только GET- или POST-методы, но и любой другой метод, например PUT или DELETE; можно использовать и нестандартный метод, например, в протоколе WebDAV используются методы COPY, PROPFIND, LOCK и др.

Старые браузеры (не поддерживавшие XMLHttpRequest) не умели делать подобные запросы. В связи с этим ряд веб-сервисов был

написан исходя из предположения, что «если метод нестандартный, то это не браузер». Некоторые веб-сервисы даже учитывали это при проверке прав доступа. Таким образом, использование современных браузеров (поддерживающих XMLHttpRequest) могло бы открыть возможности для атаки старых веб-приложений.

Разработчики стандарта CORS придумали, как именно ослабить SOP, с одной стороны, чтобы разрешить кроссдоменные запросы, а с другой — чтобы злоумышленник, воспользовавшись новым стандартом (возможностью посылать кроссдоменные Ajax-запросы с произвольными HTTP-методами), не смог сделать что-то принципиально отличное от того, что он и так мог раньше (а раньше он мог послать кроссдоменный GET-запрос, например, из тега img), и таким образом «сломать» какой-нибудь сервер, работающий по-старому стандарту и неготовый к атакам нового вида.

В отличие от простых запросов, предварительно проверяемые запросы сначала отправляют серверу в другом домене HTTP-запрос с методом OPTIONS, чтобы определить, безопасен[4] ли фактический запрос для отправки.

В частности, запрос предварительно проверяется, если выполняется любое из следующих условий:

1. Если метод не GET / POST / HEAD.

2. Кроме заголовков, которые автоматические проставляются браузером (например, Connection, User-Agent или любой другой заголовок с именем, определенным в спецификации метода Fetch в секции «Запрещенные имена заголовков (которые нельзя изменить программно)»), запрос включает заголовки, отличные от тех, которые определены спецификацией метода Fetch как «CORS-безопасные заголовки запроса», а именно:

а) Accept,

б) Accept-Language,

в) Content-Language,

г) Content-Type (но учитывайте пункт 3 ниже),

[4] Слово «безопасен» здесь понимается не так, как в RFC 7231 «Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content» (раздел 4.2.1 «Safe Methods»). Безопасность метода определяется логикой приложения.

д) Last-Event-ID,
е) DPR,
ж) Save-Data,
з) Viewport-Width,
и) Width.

3. Если заголовок Content-Type имеет значение, отличное от
а) application/x-www-form-urlencoded,
б) multipart/form-data,
в) text/plain.

4. Если зарегистрированы обработчики событий на любой объект XMLHttpRequestUpload, используемый в запросе; это достигается использованием свойства XMLHttpRequest.upload.

5. Если в запросе используется объект типа ReadableStream.

Приведем пример запроса, который будет предварительно проверен (рис. 6).

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/
post-here/';
var body = '<?xml version="1.0"?> <person> <name>Arun</name>
</person>';

function callOtherDomain(){
    if(invocation)
        {
            invocation.open('POST', url, true);
            invocation.setRequestHeader('X-PINGOTHER', 'pingpong');
            invocation.setRequestHeader('Content-Type', 'application/xml');
            invocation.onreadystatechange = handler;
            invocation.send(body);
        }
}
```

Данный запрос не является простым по двум причинам: использован нестандартный заголовок X-PINGOTHER и значение

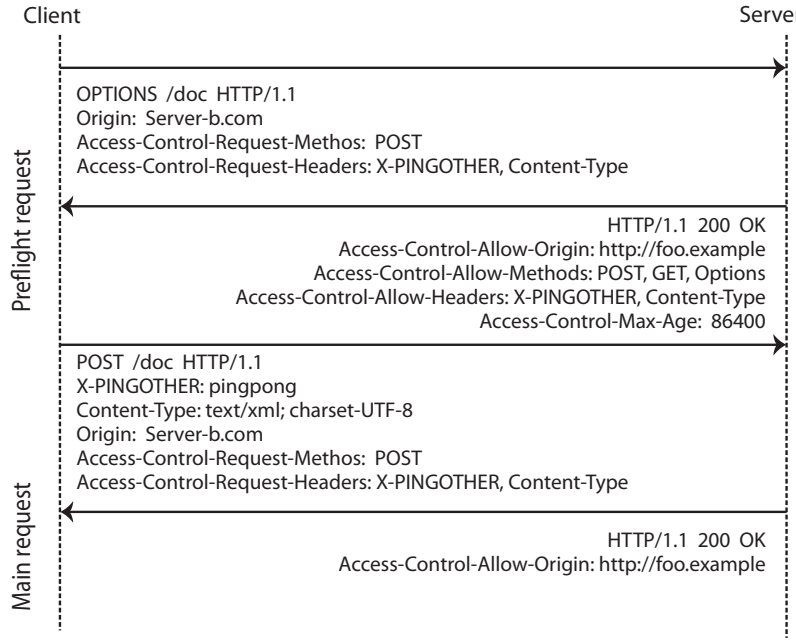

Рис. 6. Клиентские и серверные http-заголовки, реализующие CORS. Предварительно проверяемый запрос

заголовка Content-Type — application/xml (достаточно было бы и одной причины).

Посмотрим на полный обмен между клиентом и сервером. Первый обмен — предварительный запрос / ответ:

OPTIONS /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U;
Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130
Minefield/3.1b3pre
Accept: text/html, application/xhtml+xml,
application/xml; q=0.9,*/*; q=0.8
Accept-Language: en-us, en; q=0.5
Accept-Encoding: gzip, deflate

Accept-Charset: ISO-8859–1,*; q=0.7
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type

Основываясь на параметрах запроса, которые использовались в приведенном выше фрагменте JavaScript-кода, браузер понимает, что ему нужно отправить запрос с методом OPTIONS, чтобы сервер мог ответить, допустимо ли отправить POST-запрос с фактическими параметрами. Напомним, что OPTIONS — это метод HTTP/1.1, который используется для определения дополнительной информации от серверов и является безопасным[5] методом, т. е. его нельзя использовать для изменения ресурса. Обратим внимание, что вместе с запросом OPTIONS отправляются два других заголовка предварительного запроса: Access-Control-Request-Method и Access-Control-Request-Headers.

Заголовок Access-Control-Request-Method как часть предварительного запроса уведомляет сервер о том, что при отправке фактического запроса будет использован метод POST. Заголовок Access-Control-Request-Headers уведомляет сервер о том, что фактический запрос будет отправлен с пользовательскими заголовками X-PINGOTHER, а Content-Type будет иметь небезопасное значение. Сервер теперь имеет возможность определить, хочет ли он принять запрос в этих обстоятельствах.

Сервер пришлет следующий ответ:

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type

[5] Здесь безопасность понимается так, как в RFC 7231 «Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content» (раздел 4.2.1 «Safe Methods»).

Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain

Сервер возвращает список допустимых HTTP-методов, нестандартных заголовков (X-PINGOTHER) и указывает, что заголовок Content-Type может содержать небезопасные значения.

Заголовок Access-Control-Max-Age дает значение времени в секундах, в течение которого можно кэшировать ответ на запрос предварительной проверки без отправки другого запроса предварительной проверки. В этом примере сервер разрешил не посылать предварительные запросы в течение 86 400 секунд, т. е. 24 часов. Также у браузера есть максимальное внутреннее значение для времени кэширования подобных ответов, именно оно используется, если значение Access-Control-Max-Age больше.

Как только предварительный запрос завершен, отправляется реальный запрос:

POST /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html, application/xhtml+xml, application/xml; q=0.9,*/*; q=0.8
Accept-Language: en-us, en; q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859–1, utf-8; q=0.7,*; q=0.7
Connection: keep-alive
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Referer: http://foo.example/examples/preflightInvocation.html
Content-Length: 55

Origin: http://foo.example
Pragma: no-cache
Cache-Control: no-cache

<?xml version="1.0"?><person><name>Arun</name></person>

Обратим внимание, что фактический запрос POST не включает заголовки Access-Control-Request- *; они нужны только для запроса OPTIONS.

Сервер пришлет следующий ответ:

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:40 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 235
Keep-Alive: timeout=2, max=99
Connection: Keep-Alive
Content-Type: text/plain

[Some GZIP'd payload]

Подведем итог. «Непростыми» считаются все остальные запросы, например, запрос с использованием метода PUT или заголовка Authorization не подходит под вышеназванные ограничения.

Принципиальная разница между простыми и «непростыми» запросами заключается в том, что простой запрос можно сформировать и отправить на сервер и без XMLHttpRequest, например, при помощи HTML-формы.

Злоумышленник и до появления CORS мог отправить со страницы http://attacker.com произвольный GET-запрос куда угодно. Например, если создать и добавить в документ элемент <script src="http://other.domain.net">, то браузер сделает GET-запрос на этот URL.

Аналогично злоумышленник и ранее мог на своей странице создать HTML-форму и при помощи JavaScript (автоматически сгенерировать событие submit) отправить запрос с методом GET/POST и кодировкой multipart/form-data. А значит, даже старый сервер наверняка предусматривал возможность таких атак и умеет от них защищаться.

С другой стороны, запросы с нестандартными заголовками или с методами типа PUT таким образом не создать. Поэтому старый сервер может быть к ним не готов. Или, к примеру, он может предположить, что такие запросы веб-страница в принципе не умеет присылать, значит, они пришли из привилегированного приложения, — и дать им слишком много прав.

Поэтому при посылке «непростых» запросов нужно специальным образом спросить у сервера, согласен ли он в принципе на подобные кроссдоменные запросы или нет? И если сервер не ответит, что согласен, — значит, «нет».

Предварительные запросы и перенаправления

Большинство браузеров в настоящее время не разрешают перенаправление (англ. *redirect*) в ответ на предварительные запросы. Если в ответ на предварительный запрос сервер возвращает трехсотый отклик, большинство современных браузеров сообщат об ошибке. Например:

> The request was redirected to 'https://example.com/foo', which is disallowed for cross-origin requests that require preflight
>
> Request requires preflight, which is disallowed to follow cross-origin redirect

В переводе это означает, что «запрос был перенаправлен на https://example.com/foo, что запрещено для запросов из разных источников, требующих предварительной проверки».

Протокол CORS первоначально требовал такого поведения, но впоследствии правила были изменены. Тем не менее не все браузеры реализовали это изменение.

В связи с этим пока браузеры не будут приведены в соответствие со спецификацией, можно обойти это ограничение, выполнив одно (или оба) из следующих действий:

— изменить поведение сервера, чтобы избежать предварительной проверки и/или избежать перенаправления, если имеется контроль над сервером, на который делается запрос;

— изменить запрос так, чтобы он стал простым и не потребовал предварительной проверки.

Если невозможно внести эти изменения, то нужно выбрать другой способ:

1) сделать простой запрос, чтобы определить (используя Response.url для Fetch API или XHR.responseURL), по какому URL-адресу будет в действительности направлен предварительный запрос;

2) сделать еще один запрос («настоящий» запрос), используя URL-адрес, полученный из Response.url или XMLHttpRequest.responseURL на первом этапе.

Однако если запрос инициирует предварительную проверку из-за присутствия в нем заголовка Authorization, то не удастся обойти данную проблему, используя описанные выше шаги, и вообще не удастся ее решить, если нет контроля над сервером, на который делается запрос.

Запросы с учетными данными

Наиболее интересная возможность, имеющаяся как у XMLHttpRequest, так и Fetch, а также в CORS, — это возможность делать предварительные запросы, которые имеют доступ к файлам cookie и информации HTTP-аутентификации. По умолчанию в межсайтовых вызовах XMLHttpRequest или Fetch браузеры не отправляют учетные данные. Для отправки учетных данных должен быть установлен конкретный флаг для объекта XMLHttpRequest или конструктора Request при его вызове.

Пример. Скрипт, изначально загруженный с http://foo.example, выполняет простой GET-запрос к ресурсу http://bar.other и требует отправки cookies (рис. 7). Содержимое на foo.example может иметь такой JavaScript-код:

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/credentialed-content/';

function callOtherDomain(){
    if(invocation) {
        invocation.open('GET', url, true);
        invocation.withCredentials = true;
        invocation.onreadystatechange = hadler;
        invocation.send();
    }
}
```

Пример обмена между клиентом и сервером:

```
GET /resources/access-control-with-credentials/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U;
Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130
Minefield/3.1b3pre
Accept: text/html, application/xhtml+xml,
application/xml; q=0.9,*/*; q=0.8
Accept-Language: en-us, en; q=0.5
```

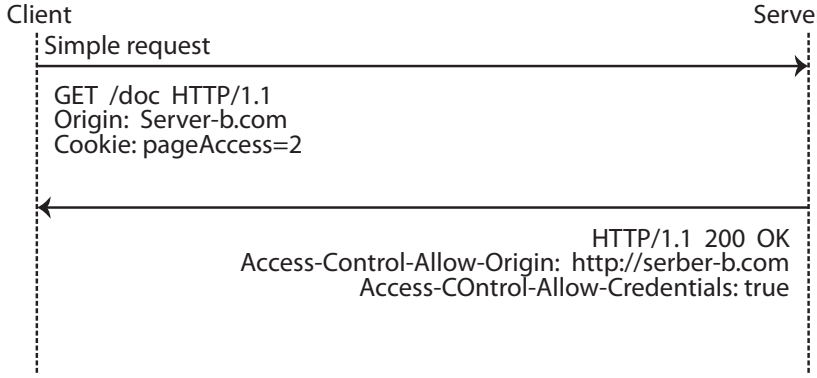

Рис. 7. Клиентские и серверные http-заголовки, реализующие CORS. Отправка cookie

Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859–1,*; q=0.7
Connection: keep-alive
Referer: http://foo.example/examples/
credential.html
Origin: http://foo.example
Cookie: pageAccess=IAmAuthorized

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:34:52 GMT
Server: Apache/2.0.61 (Unix) PHP/4.4.7 mod_ssl/2.0.61 OpenSSL/0.9.7e
mod_fastcgi/2.4.2 DAV/2 SVN/1.4.2
X-Powered-By: PHP/5.2.6
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Pragma: no-cache
Set-Cookie: pageAccess=OK; expires=Wed,
31-Dec-2018 01:34:53 GMT
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 106
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain

[text/plain payload]

Несмотря на то что запрос содержит заголовок cookie с авторизационными данными, предназначенный для сервера http://bar.other, если bar.other не пришлет заголовок Access-Control-Allow-Credentials: true, ответ будет проигнорирован браузером и не будет доступен JavaScript-коду.

При ответе на запрос, содержащий учетные данные, сервер должен указать конкретный источник в значении заголовка Access-Control-Allow-Origin, а не подстановочный знак «*».

Для практического освоения изложенного материала рекомендуем провести эксперименты, исходные коды на JavaScript приведены в конце главы.

3.5. Методы обхода и ослабления Same Origin Policy

В некоторых случаях жесткие ограничения Same Origin Policy создают проблемы для крупных веб-сайтов, которые используют несколько поддоменов, например, e.mail.ru, news.mail.ru, games.mail.ru и т. д. Современные браузеры поддерживают ряд методов для ослабления требований Same Origin Policy.

Свойство document.domain

Страница может изменить свой origin с некоторыми ограничениями. JavaScript сценарий может изменить текущее значение document.domain на значение супердомена; в этом случае более короткий домен используется для последующих проверок происхождения. Также сценарий может присвоить document.domain значение текущего домена (зачем это нужно — описано ниже). А вот установить для document.domain значение othercompany.com не удастся, поскольку это не супердомен company.com.

П р и м е р. Пусть скрипт выполняется в документе, загруженном с URL-адреса http://store.company.com/dir/other.html, и выполняет следующее присваивание:

```
document.domain = "company.com";
```

После этого присваивания страница сможет пройти проверку на равенство источников (origin) со страницей http://company.com/dir/page.html. Важно отметить, что для успешного прохождения этой проверки необходимо, чтобы на странице http://company.com/dir/page.html тоже было установлено

```
document.domain = "company.com";
```

Казалось бы, это присваивание ничего не меняет, но на самом деле при этом присваивании port сбрасывается в null.

Изменения свойства document.domain используются, например, для разрешения взаимодействия родительского окна и дочернего iframe. В примере ниже создано два документа, которые загружаются в основное окно и в iframe с адресов test.example.com и example.com соответственно.

Файл, получаемый с test.example.com:

```
<! DOCTYPE HTML>
<html>
<head>
   <meta charset="utf-8">
   <script>
      function changeOrigin(){
      console.log(document.domain);
      document.domain = 'example.com';
      console.log(document.domain);
      }

      function toFrame(){
      var iframe = document.getElementById('myFrame');
      try {
       console.log(iframe.contentWindow.document.body.innerHTML);
      } catch (e) {
      console.log("Ошибка: " + e.message);
      }
      }
   </script>
</head>
<body onload='changeOrigin()'>
   <iframe src='http://example.com' id='myFrame'>       </iframe>
   <input type='button' value='Get content' onclick='toFrame()'>
</body>
</html>
```

Файл, получаемый с test.example.com:

```
<! DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8">
<script>

    function changeOrigin(){
        console.log(document.domain);
        document.domain = 'example.com';
        console.log(document.domain);
}
</script>
</head>
<body onload='changeOrigin()'>
<p>Hello from frame!</p>
</body>
</html>
```

Обратим внимание на то, что присваивание document.domain = ‘example.com’ происходит в двух документах, в том числе и в документе, который загружен с example.com. Легко проверить, что без этого браузер запретит получение контента iframe и выдаст ошибку.

Чтобы у читателей не возникало путаницы, отметим, что document.domain используется для взаимодействия окон и фреймов. Посылка кроссдоменных Ajax-запросов на удаленные серверы с помощью document.domain не разрешается — для этого применяется CORS. Действительно, злоумышленник, посылающий кроссдоменный Ajax-запрос в скрипте, всегда может присвоить в document.domain нужное значение. Здесь требуется именно разрешение от сервера.

Свойство window.location

Какие именно ограничения накладывает Same Origin Policy на взаимодействие скриптов, загруженных в разные окна, нужно изучать по документации. Не все действия запрещены: окна могут

менять location друг друга, даже если они из разных источников. А вот считать значение свойства location нельзя, так как это раскрывает конфиденциальные данные пользователя.

```
<iframe src="https://example.com"></iframe>

<script>
    var iframe = document.body.children[0];
    iframe.onload = function() {
        try {
            // не сработает (чтение)
            alert(iframe.contentWindow.location.href);
        } catch (e) {
        alert("Ошибка при чтении: " + e.message);
    }
        // сработает (запись)
        iframe.contentWindow.location.href = 'https://wikipedia.org';
        iframe.onload = null;
    }
</script>
```

Общение окон с разных доменов: Window.postMessage()

Обычно скрипты, находящиеся в разных окнах (в том числе всплывающих окнах и фреймах), могут получать доступ друг к другу, только если они имеют один источник, т. е. используется один и тот же протокол, номер порта и хост. Метод window.postMessage (если используется правильно) предоставляет управляемый механизм для безопасного обхода этого ограничения.

Метод window.postMessage — это реализация web messaging или cross-document messaging, разработанного WHATWG HTML5 API, который представлен в черновой спецификации, позволяющей документам из разных источников взаимодействовать друг с другом в среде браузера.

В целом одно окно может получить ссылку на другое (например, через targetWindow = window.opener), а затем отправить ему MessageEvent с помощью targetWindow.postMessage(). Получающее окно

может при необходимости обрабатывать это событие. Аргументы, передаваемые в window.postMessage (т. е. в сообщение), доступны принимающему окну через соответствующий объект события.

Метод имеет следующий синтаксис:

```
targetWindow.postMessage(message,
tagetOrigin, [transfer]);
```

targetWindow — ссылка на окно, которое получит сообщение. Существуют следующие способы получения такой ссылки:

— window.open (чтобы создать новое окно и затем сослаться на него);

— window.opener (для ссылки на окно, которое создало данное окно);

— HTMLIFrameElement.contentWindow (для ссылки на встроенный <iframe> из его родительского окна);

— window.parent (для ссылки на родительское окно из встроенного <iframe>);

— window.frames + значение индекса (именованное или числовое).

message — данные для отправки в другое окно. Данные сериализуются с использованием алгоритма структурированного клонирования. Это означает, что можно безопасно передавать широкий спектр объектов данных в окно назначения без необходимости их сериализации.

targetOrigin — источник[6] (origin) для targetWindow, он должен быть таким, чтобы браузер передал документу, загруженному в targetWindow, событие о том, что было отправлено сообщение. Источник должен либо быть равен «*» (без ограничений на источник), либо быть задан в виде URL-адреса.

Скрипт — отправитель сообщения, вообще говоря, не знает, какой документ сейчас загружен в iframe или дочернее всплывающее окно, известен только id окна. Возможно, что в iframe уже загрузил-

[6] Источник здесь надо трактовать как origin — упорядоченная тройка из протокола, хоста и порта, т. е. источник, откуда был загружен документ, находящийся в окне с идентификатором targetWindow. Речь не идет об источнике сообщения, т. е. об отправителе сообщения.

ся зловредный сайт http://evil.com и зарегистрировал обработчик события, связанного с получением сообщения:

```
window.addEventListener("message",
receiveMessage, false);
```

Если во время отправки сообщения браузер выяснит, что протокол, имя хоста или порт документа, загруженного в targetWindow, не совпадают с указанными в targetOrigin, сообщение не будет отправлено. Браузер сгенерирует событие, несущее в себе сообщение, в том и только в том случае, когда origin документа, загруженного в targetWindow, будет совпадать с данными, указанными в targetOrigin.

Этот механизм обеспечивает контроль над тем, куда отправляются сообщения: например, если postMessage использовался для передачи пароля, то этот аргумент должен быть задан как URL и origin предполагаемого получателя должен соответствовать данному URL. Это позволит предотвратить перехват пароля злоумышленником. Нужно всегда указывать конкретный targetOrigin, а не *, если известно, кто должен быть получателем сообщения.

П р и м е р. Предположим, мы хотим, чтобы документ *A*, загруженный с example.net, связывался с документом *B*, загруженным в iframe или всплывающее окно с example.com. JavaScript-код для документа *A* будет выглядеть следующим образом:

```
let o = document.
getElementsByTagName('iframe')[0];
o.contentWindow.postMessage('Hello B', 'http://example.com/');
```

JavaScript-код для документа *B* будет выглядеть следующим образом:

```
function receiver(event) {
    if (event.origin == 'http://example.net') {
        if (event.data == 'Hello B') {
```

```
            event.source.postMessage('Hello A, how are you?', event.origin);
        }
        else {
            alert(event.data);
        }
    }
}
window.addEventListener('message', receier, false);
```

Обработчик событий настроен на получение сообщений из документа *A*. Используя свойство origin, он затем проверяет, соответствует ли origin отправителя ожидаемому (в данном примере http://example.net). Затем документ *B* просматривает сообщение и либо отображает его пользователю, либо посылает собственное сообщение документу *A*.

Напомним, что метод addEventListener используется для назначения обработчика события и принимает три параметра: название события (например, click, submit или, как в данном примере, message), имя функции-обработчика и фазу, на которой происходит обработка события.

Неграмотное использование postMessage может сделать приложение уязвимым к атакам DOM XSS.

Пример:

```
//Listener on http://www.example.com/
window.addEventListener("message",
function(message){
if(message.origin == "http://www.example.com"){
    document.getElementById("message").innerHTML = message.data;
}
});
```

Необходимо проводить проверку безопасности приложений с использованием автоматических средств тестирования.

JSON with padding

Так как Same Origin Policy не запрещает загрузку JavaScript-кода из другого источника через тег <script>, то появляется возможность для кроссдоменной передачи данных.

Пользователь загрузил документ с http://domanin1.com. Соответственно, JavaScript-код с Origin, равным <http, domanin1.com, 80>, может отправлять Ajax-запросы на сервер http://domanin1.com, а вот посылка запросов на http://domanin2.com запрещена Same Origin Policy (если не был настроен CORS), что показано на рис. 8.

Пользовательский скрипт желает загрузить данные про покупателя с идентификатором № 13 с адреса http://domanin2.com/customer/13. Для этого необходимо динамически создать элемент <script> с нужным значением атрибута src=’нужный URL’ и добавить его как дочерний элемент в <head>. Браузер при этом автоматически запустит процесс загрузки данных с нужного URL, отправив на сервер domain2.com HTTP запрос с cookie для domain2.com.

```
function addScript(src) {
    var elem = docment.createElement("script");
    elem.src = src;
    document.head.appendChild(elem);
}

addScript(' http://domanin2.com/customer/13');
```

Отметим, что ответы содержат не JSON, а некоторый JavaScript-код, и обрабатываются они интерпретатором JavaScript, а не парсером JSON. К чему это приводит?

Предположим, сервер вернул данные в JSON-формате:

```
{"Name": "Adam", "Id": 13, "Rank": 1}
```

В свою очередь, браузер получит ответ, разберет его содержимое, интерпретирует сырые JSON-данные как блок и вернет ошибку синтаксиса. Даже если данные были интерпретированы как литеральный объект JavaScript, к нему невозможно получить доступ

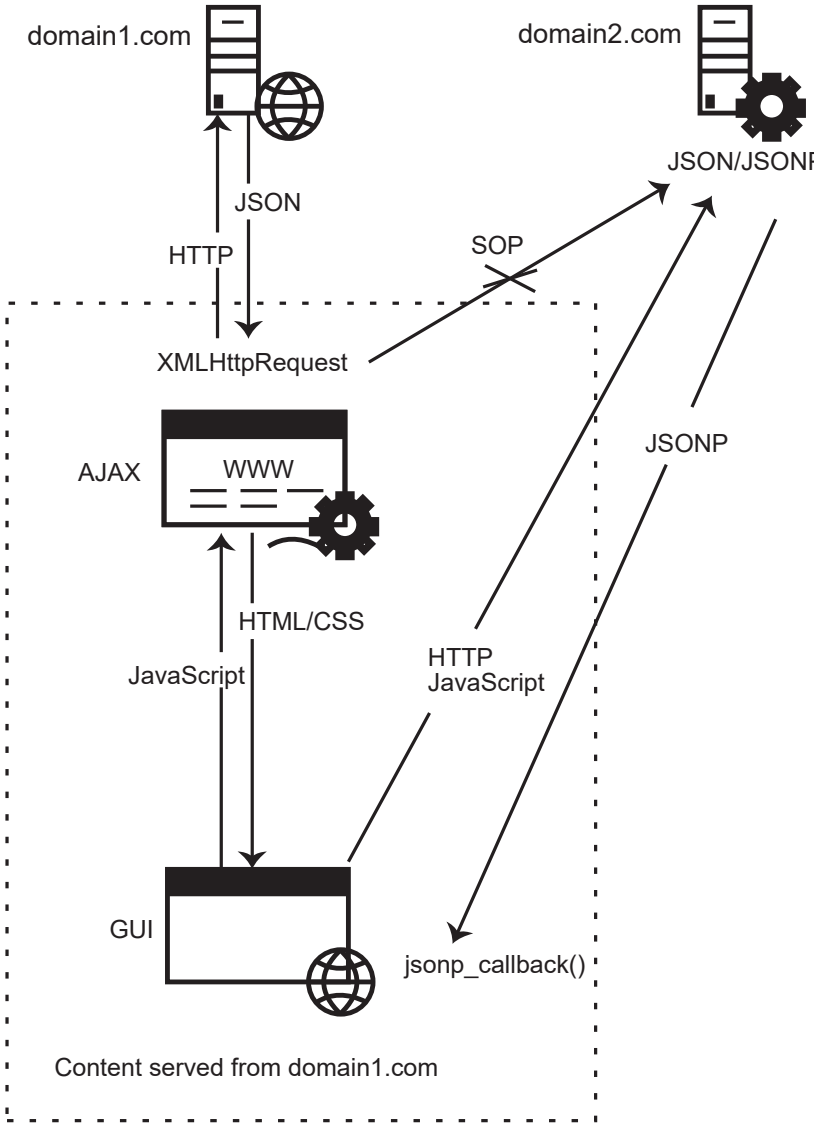

Рис. 8. Технология JSONP как метод обхода SOP

из JavaScript, выполняемого в браузере, поскольку без присвоения переменной объектные литералы уничтожаются сборщиком мусора.

Решение очень простое. Вместе с запросом клиента отдельным параметром передается название функции; обычно такой параметр называется callback. Например:

```
addScript('http://domanin2.com/customer/13
&callback=onUserData');
```

Сервер кодирует данные в JSON и оборачивает их в вызов функции, название которой получает из параметра callback:

```
onUserData({"Name": "Adam", "Id": 13, "Rank": 1});
```

Если другой источник будет скомпрометирован, то в ответе может быть любой JavaScript-код, в том числе деструктивного характера.

Рекомендуем читателю ознакомиться со спецификацией на Cross-Origin Resource Sharing [2]. Сущность технологии Cross-Origin Resource Sharing хорошо разъясняется в [3].

3.6. Дополнение. Листинги с исходным кодом

Для проведения экспериментов с CORS необходимо несколько веб-серверов с разными источниками (origin). Простейший способ смоделировать такую ситуацию — запустить два сервера на разных TCP-портах.

Для запуска веб-серверов используем стандартный пакет http для Node.js. Первый сервер запускаем на 127.0.0.1:3000, второй — на 127.0.0.1:4200. С первого сервера получаем файл index.html и фиксируем origin <http, localhost, 3000>, запросы посылаем на API-сервер с другим origin.

Файл server3000.js

```
var http = require('http');
var fs = require('fs');
```

```
var server = new http.Server();
server.listen(3000, '127.0.0.1');

server.on('request', function(req, res){
    res.setHeader('content-type', 'text/html');.
    fs.readFile("index.html", {encoding: 'utf-8'}, function(err, data){
        if(err){
            console.error(err.message);
        }else{
            res.end(data);
        }
    });
});
```

Файл index.html

Обратим внимание на то, что кроссдоменные запросы из тегов link и img не требуют использования CORS. Перезагрузка картинки в результате работы функции changeImg также не требует CORS. А вот работоспособность функции getContent принципиально зависит от CORS-заголовков, выставленных API-сервером.

```
<! DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8">
    <link href='http://solod.webhostapp.com/style.css' rel='stylesheet'
    type='text/css'>
    <script>
    function changeImg(){
    document.getElementById('pic1').src =
    'http://localhost:4200/otherimg';
    }

    function getContent(){
    xhr = new XMLHttpRequest();
    xhr.open('GET', 'http://localhost:4200/content');
```

```
    xhr.onload = xhr.onerror = function() {
        if (this.status == 200) {
            console.log(this.response);
        } else {
            console.log("error " + this.status);
        };
    };
    xhr.send();
    }
    </script>
</head>
<body>
    <p><img src='https://upload.wikimedia.org /wikipedia/diagram.
png' alt='CSP'></p>
    <p><img id='pic1' src='http://localhost:4200/img' alt='foto'></p>
    <input type='button' value='change img'
onclick='changeImg()'><br>
    <input type='button' value='get content' onclick='getContent()'>
</body>
</html>
```

Файл server4200.js

В качестве эксперимента надо удалить отсылку CORS-заголовков и выяснить, какие ошибки возникнут на стороне получателя.

```
var http = require('http');
var fs = require('fs');
var url = require('url');

var server = new http.Server();
server.listen(4200, '127.0.0.1');

server.on('request', function(req, res){
    var urlParsed = url.parse(req.url, true);
    console.log(urlParsed);
    console.log(req.headers);
```

```
    if(urlParsed.pathname == '/img'){
        fs.readFile("img.jpg", function(err, data){
        if(err){
        console.error(err.message);
        }else{
        res.end(data);
        }
    });
 };
    if(urlParsed.pathname == '/otherimg'){
        fs.readFile("otherimg.jpg", function (err, data){
        if(err){
        console.error(err.message);
        }else{
            res.end(data);
        }
    });
};

if(urlParsed.pathname == '/content'){
    fs.readFile("content.txt", {encoding: 'utf-8'}, function(err, data){
        if(err){
            con-sole.error(err.message);
        }else{
            res.setHeader("Access-Control-Allow-Origin",
"http://localhost:3000");
            res.setHeader("Access-Control-Allow-Credentials", "true");
            res.setHeader("Access-Control-Allow-Headers",""Origin,
Content-Type, X-Auth-Token, Authorization");
            res.end(data);
            }
        });
    };
});
```

Отметим, что сменить источник для файла, загруженного с сервера localhost:3000, не удастся простым вызовом функции. Например:

```
function changeOrigin(){
    document.domain = 'example.com';
}
```

Действительно, тогда бы любой скрипт, выполняющий кросс-доменный Ajax-запрос, предварительно «настраивал» бы origin нужным образом. Смена document.domain позволяет передавать данные, например, из родительского окна в iframe, и наоборот.

Контрольные вопросы

1. Как известно, существует HTTP-заголовок Referer, в котором обычно указан адрес страницы, с которой инициирован запрос. Например, при отправке XMLHttpRequest со страницы http://example.com/path/to/file на http://other.net заголовки могут быть такими:

```
Accept: */*
Accept-Charset: windows-1251,*; q=0.3
Accept-Encoding: gzip, deflate
Accept-Language: ru-RU, ru; q=0.8, en-US; q=0.6, en; q=0.4
Connection: keep-alive
Host: other.net
Origin: http://example.com
Referer: http://example.com/path/to/file
```

Как видно, здесь присутствуют и Referer и Origin. Зачем нужен Origin, если Referer содержит даже более полную информацию? Может ли заголовок Referer отсутствовать или содержать неверное значение?

2. Используя window.postMessage, разработчик посылает сообщения, а в другом окне делает проверку источника (origin), с которого пришло сообщение. Разработчик хочет получать сообщения только от http://www.example.com. Нет ли ошибки в проверке источника? Если есть, то какая?

```
//Listener on http://www.examplereceiver.com/
window.addEventListener("message", function(message){
if(/^http://www.example.com$/.test(message.origin)){
        console.log(message.data);
    }
});
```

Ответы

1. Origin нужен потому, что Referer передается не всегда. Например:

— при запросе с HTTPS на HTTP нет заголовка Referer;

— политика Content Security Policy может запрещать пересылку Referer;

— по стандарту Referer является необязательным HTTP-заголовком, в некоторых браузерах есть настройки, которые запрещают его посылать.

Именно потому, что на Referer полагаться нельзя, был придуман заголовок Origin, который гарантированно присылается при кроссдоменных запросах. Что же касается «неправильного» Referer, то такая ситуация практически невозможна. Много лет назад в браузерах были ошибки, которые позволяли подменить Referer из JavaScript, но они давно исправлены. В настоящее время Referer подменить нельзя.

2. Разрешены не только сообщения от «www.examplesender.com», но также «wwwaexamplesender.com», «wwwbexamplesender.com» и т. д.

Глава 4
ВЕБ-СЕРВИСЫ

Веб-сервис — идентифицируемая URL-адресом программная система со стандартизированными интерфейсами. Веб-службы могут взаимодействовать друг с другом и со сторонними приложениями посредством сообщений, основанных на определенных протоколах (SOAP, XML-RPC и т. д.) и соглашениях (REST). Веб-служба является единицей модульности при использовании сервис-ориентированной архитектуры приложения.

Ответим на вопрос «Зачем нужны веб-сервисы?» на примере задачи о предоставлении сведений о вылетах самолетов из аэропорта. Можно создать веб-приложение — онлайн-табло с красивым пользовательским интерфейсом, конечные пользователи (люди) будут просматривать страницы этого приложения и получать нужную информацию. Если же потребителем данных о вылетах выступает не человек, а другое приложение, нужен ли ему красивый интерфейс? Не нужен, так как вычленять данные из html-страницы — это тяжелая задача; осложняет ситуациию и то, что дизайн страницы, html-разметка могут меняться, и парсер придется регулярно переписывать. Приложению, которое заинтересовано в данных о вылетах, предпочтительно получать эти данные в стандартизированном, легко разбираемом формате, таком как XML или JSON.

Итог вышесказанного: веб-приложения — для людей, веб-сервисы — для машин. По сути, веб-сервисы — это реализация абсолютно четких интерфейсов обмена данными между различными

приложениями, которые могут быть не только написаны на разных языках, но и распределены на разных узлах Сети.

Работу с веб-сервисами проиллюстрируем на примере создания веб-сервиса, который вычисляет индекс массы тела[7] (ИМТ). Будем использовать стандарты SOAP и WSDL. Создадим три файла: сервер, клиент и WSDL-файл.

Сервер, реализующий веб-сервис, создадим на Node.js. В этом файле будет определен удаленный веб-сервис, к которому могут обращаться клиенты, предоставляя аргументы для расчета ИМТ. Ответ возвращается вызывающему клиенту.

```
var soap = require('soap');
var express = require('express');
var app = express();

var service = {
BMI_Service : {
    BMI_Port :{
        calculateBMI: function(args){
            var n = (args.weight)/
                    (args.height*args.height);
            console.log(n);
            return {bmi: n};
            }
        }
    }
}
var xml = require('fs').readFileSync ('./bmicalc.wsdl','utf8');
var server = app.listen(3030, function(){
    var host = "127.0.0.1";
    var port = server.address().port;
});
soap.listen(server,'/bmicalc', service, xml);
```

[7] Индекс массы тела определяют путем деления веса (в килограммах) на числовое выражение роста (в метрах), возведенное в квадрат.

Далее создадим клиента, который будет использовать SOAP-сервис, определенный в файле server.js. Запрос, инициированный в файле client.js, предоставит аргументы для SOAP-службы. Запрос будет послан на соответствующий URL-адрес с портами и конечными точками службы SOAP.

```
var express = require('express');
var soap = require('soap');
var url = "http://localhost:3030/bmicalc?wsdl";
var args = {weight:65.7, height:1.63};
soap.createClient(url, function(err, client){
    if(err)
        console.error(err);
    else {
        client.calculateBMI(args,
            function(err, response){
            if(err)
            console.error(err);
            else {
                    console.log(response);
            res.send(response);
        }
        })
    }
});
```

Далее создадим файл wsdl — это основанный на XML формат описания способа обмена данными; он определяет, как получить доступ к удаленному веб-сервису. Назовем wsdl-файл bmicalc.wsdl.

```
<definitions name="HelloService"
targetNamespace = "http://www.examples.com/wsdl/HelloService.wsdl"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns: soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns: tns="http://www.examples.com/wsdl/HelloService.wsdl"
xmlns: xsd="http://www.w3.org/2001/XMLSchema">
```

```
<message name="getBMIRequest">
<part name="weight" type="xsd: float"/>
<part name="height" type="xsd: float"/>
</message>

<message name="getBMIResponse">
<part name="bmi" type="xsd: float"/>
</message>

<portType name="Hello_PortType">
<operation name="calculateBMI">
<input message="tns: getBMIRequest"/>
<output message="tns: getBMIResponse"/>
</operation>
</portType>

<binding name="Hello_Binding" type="tns: Hello_PortType">
<soap: binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="calculateBMI">
<soap: operation soapAction="calculateBMI"/>
<input>
<soap: body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn: examples: helloservice"
use="encoded"/>
</input>
<output>
<soap: body
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn: examples: helloservice"
use="encoded"/>
</output>
</operation>
</binding>
```

```
<service name="BMI_Service">
<documentation>WSDL File for HelloService </documentation>
<port binding="tns: Hello_Binding" name="BMI_Port">
<soap: address
location="http://localhost:3030/bmicalc/" />
</port>
</service>
</definitions>
```

Контрольные вопросы

1. В чем разница между сервис-ориентированной архитектурой (англ. *service-oriented architecture*) и веб-сервисами?

2. В чем разница между REST и SOAP веб-сервисами? В чем преимущества первого и второго?

3. Объясните понятие WSDL.

4. Возможно ли посылать SOAP-сообщения с вложением?

Глава 5
ФРЕЙМВОРК EXPRESS

5.1. Быстрый старт

Рассмотрим процесс создание веб-приложения с помощью фреймворка Express. Возникает вопрос: зачем нужен дополнительный фреймворк, если можно воспользоваться модулем http, который есть в Node.js API? Express, который сам использует модуль http, предоставляет ряд готовых абстракций, упрощающих разработку веб-приложения, например, таких: обработка отправленных форм, работа с cookie, CORS и т. д. [4–6].

Express — фреймворк web-приложений для Node.js, реализованный как свободное и открытое программное обеспечение под лицензией MIT (лицензия открытого программного обеспечения, разработанная Массачусетским технологическим институтом). Он спроектирован для создания как веб-приложений, так и API. Первая версия фреймворка была выпущена в ноябре 2010 г.

Рассмотрим пример создания простейшего веб-приложения.

Создадим для проекта новый каталог, который назовем, к примеру, expressapp. Для хранения информации обо всех зависимостях проекта определим в этом каталоге новый файл package.json:

```
{
    "name": "expressapp",
    "version": "1.0.0",
```

```
    "dependencies": {
        "express": "^4.16.4"
    }
}
```

Далее перейдем к этому каталогу в командной строке и для добавления всех нужных пакетов выполним команду:

```
npm install
```

Создадим в каталоге проекта новый файл app.js со следующим содержанием:

```
// подключаем express
const express = require("express");
// создаем объект приложения
const app = express();
// определяем обработчик для маршрута "/"
app.get("/", function(req, res){
    // отправляем ответ
    res.send("<h2>Привет Express!</h2>");
});
// прослушиваем подключения на 3000 порту
app.listen(3000);
```

Поясним пример. Для использования Express необходимо, во-первых, создать объект, который будет представлять приложение:

```
const app = express();
```

Цикл работы Express-приложения состоит в получении пользовательского запроса, его обработке (как правило, многоступенчатой) и послылке ответа. Для обработки запросов в Express определен ряд встроенных функций, одной из которых является функция app.get(). Она обрабатывает GET-запросы протокола HTTP и по-

зволяет связать маршруты[8] с определенными обработчиками. Для этого в функцию первым параметром передается маршрут, вторым — обработчик, который будет вызываться, если запрос к серверу соответствует данному маршруту:

```
app.get("/", function(request, response){
    // отправляем ответ
    res.send("<h2>Привет Express!</h2>");
});
```

Для проверки работоспособности надо запустить проект и обратиться в браузере по адресу http://localhost:3000/.

Когда фреймворк Express получает запрос, этот запрос передается в конвейер обработки. Конвейер состоит из набора функций промежуточной обработки (англ. *middleware*), которые получают данные запроса и ответа, обрабатывают их и либо передают следующим функциям промежуточной обработки, либо заканчивают цикл обработки, например, отсылая ответ клиенту.

Функции промежуточной обработки могут выполнять следующие задачи:

1) выполнение любого кода;

2) внесение изменений в объекты запросов и ответов;

3) завершение цикла «запрос–ответ»;

4) вызов следующей функции промежуточной обработки из стека.

Если текущая функция промежуточной обработки не завершает цикл «запрос–ответ», она должна вызвать функцию next() для передачи управления следующей функции промежуточной обработки. В противном случае обработка запроса зависнет.

В предыдущем примере обработка состояла в вызове app.get(). Сначала сравнивался запрошенный адрес «/», и если между адресом и маршрутом было соответствие, то клиенту посылалась строка «<h2>Привет Express!</h2>».

---

[8] На данном этапе под маршрутом можно понимать URL-адрес или класс похожих арресов, например, все адреса, начинающиеся на http://localhost:3000/products.

При необходимости можно встроить в конвейер обработки запроса на любом этапе любую функцию middleware. Для этого применяется метод app.use(). Изменим файл app.js, например, следующим образом:

```
const express = require("express");
const app = express();
app.use(function(request, response, next){
    console.log("Middleware 1");
    next();
});
app.use(function(request, response, next){
    console.log("Middleware 2");
    next();
});
app.get("/", function(req, res){
    res.send("<h2>Привет Express!</h2>");
});
app.listen(3000);
```

Функция, которая передается в app.use(), принимает три параметра:

1) request — данные запроса;

2) response — объект для управления ответом;

3) next — следующая функция в конвейере обработки.

Каждая из функций middleware просто выводит на консоль сообщение и в конце вызывает следующую функцию с помощью next().

Middleware помогают выполнять отдельные задачи, которые должны быть сделаны до отправки ответа. Стандартная задача — логгирование запросов. Приведем пример соответствующего кода:

```
const express = require("express");
const fs = require("fs");

const app = express();
app.use(function(req, res, next){
```

```
    let now = new Date();
    let hour = now.getHours();
    let minutes = now.getMinutes();
    let seconds = now.getSeconds();
    let data = `${hour}:${minutes}:${seconds} ${req.method} ${req.url}`;
    console.log(data);
    fs.appendFile("server.log", data + "\n",
        function(){});
    next();
});

app.get("/", function(req, res){
    res.send("Hello");
});
app.listen(3000);
```

5.2. Маршрутизация

В зависимости от URL-адреса, по которому обратился пользователь, обработка запроса должна производиться по-разному. Маршрутизация определяет, как приложение отвечает на клиентский запрос к конкретному URL-адресу. В приложении определяются маршруты, а также обработчики этих маршрутов. Если запрос соответствует определенному маршруту, то вызывается соответствующий обработчик.

Маршруты — это часть кода Express, связывающая HTTP-действия (GET, POST, PUT, DELETE и т. д.), URL пути (шаблона) и функцию, которая обрабатывает этот шаблон. Приведенный ниже код служит примером простейшего использования маршрутов. Отметим, что запросы, которые не соответствуют корневому маршруту «/», обрабатываться не будут.

```
var express = require('express');
var app = express();
app.get('/', function(req, res) {
```

```
    res.send('hello world');
});
```

В зависимости от HTTP-метода, использованного в запросе, обработка может производиться по-разному. Express различает следующие методы маршрутизации: get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, m-search, notify, subscribe, unsubscribe, patch, search и connect.

Пример:

```
// GET method route
app.get('/', function (req, res) {
    res.send('GET request to the homepage');
});

// POST method route
app.post('/', function (req, res) {
    res.send('POST request to the homepage');
});
```

Существует особый метод маршрутизации — app.all(), не являющийся производным от какого-либо метода HTTP. Этот метод используется для загрузки функций промежуточной обработки в пути для всех методов запросов.

Пути маршрутов и регулярные выражения

Пути маршрутов в сочетании с методом запроса определяют конкретные адреса (конечные точки), в которых могут быть созданы запросы. Пути маршрутов могут представлять собой строки, шаблоны строк или регулярные выражения.

Приведем примеры путей маршрутов на основе строк. В этом случае путь маршрута соответствует запросам, в которых URL имеет часть path, равную строке /about:

```
app.get('/about', function (req, res) {
    res.send('about');
});
```

Аналогично для /random.text:

```
app.get('/random.text', function (req, res) {
    res.send('random.text');
});
```

Приведем примеры путей маршрутов на основе шаблонов строк. Здесь путь маршрута соответствует запросам, у которых URL имеет часть path, начинающуюся с «ab» (после чего возможен один произвольный символ) и заканчивающуюся на «cd»:

```
app.get('/ab?cd', function(req, res) {
    res.send('ab?cd');
});
```

Аналогично для строки '/ab*cd', за исключением того, что между «ab» и «cd» может находиться произвольная подстрока:

```
app.get('/ab*cd', function(req, res) {
    res.send('ab*cd');
});
```

Приведем примеры путей маршрутов на основе регулярных выражений. Здесь путь маршрута соответствует запросам, у которых часть path содержит символ «a»:

```
app.get(/a/, function(req, res) {
    res.send('/a/');
});
```

Следующий путь маршрута соответствует запросам, у которых часть path заканчивается на «fly», например, butterfly, dragonfly, но не butterflyman:

```
app.get(/.*fly$/, function(req, res) {
    res.send('/.*fly$/');
});
```

Обработчики маршрутов

Для обработки запроса можно указать несколько функций обратного вызова, подобных middleware. Единственным исключением является то, что эти обратные вызовы могут инициировать next('route') для обхода остальных обратных вызовов маршрута. С помощью этого механизма можно включить в маршрут предварительные условия, а затем передать управление последующим маршрутам, если продолжать работу с текущим маршрутом не нужно.

Обработчики маршрутов могут принимать форму функции, массива функций или их сочетания, как показано в следующих примерах.

Одна функция обратного вызова может обрабатывать один маршрут. Например:

```
app.get('/example/a', function (req, res) {
    res.send('Hello from A!');
});
```

Один маршрут может обрабатываться несколькими функциями обратного вызова (обязательно следует указывать объект next). Например:

```
app.get('/example/b', function (req, res, next) {
    console.log('the response will be sent by the next function …');
    next();
}, function (req, res) {
    res.send('Hello from B!');
});
```

Массив функций обратного вызова может обрабатывать один маршрут. Например:

```
var cb0 = function (req, res, next) {
    console.log('CB0');
    next();
}
var cb1 = function (req, res, next) {
    console.log('CB1');
    next();
}
var cb2 = function (req, res) {
    res.send('Hello from C!');
}
app.get('/example/c', [cb0, cb1, cb2]);
```

Маршрут может обрабатываться сочетанием независимых функций и массивов функций. Например:

```
var cb0 = function (req, res, next) {
    console.log('CB0');
    next();
}
var cb1 = function (req, res, next) {
    console.log('CB1');
    next();
}
app.get('/example/d', [cb0, cb1], function (req, res, next) {
    console.log('the response will be sent by the next function …');
    next();
}, function (req, res) {
    res.send('Hello from D!');
});
```

Методы ответа

Методы объекта ответа, перечисленные в табл. 2, могут передавать ответ клиенту и завершать цикл «запрос–ответ». Если ни один из этих методов не будет вызван из обработчика маршрута, обработка клиентского запроса зависнет.

Таблица 2

Методы объекта ответа

Метод	Описание
res.download()	Приглашение к загрузке файла
res.end()	Завершение процесса ответа
res.json()	Отправка ответа JSON
res.jsonp()	Отправка ответа JSON с поддержкой JSONP
res.redirect()	Перенаправление ответа
res.render()	Вывод шаблона представления
res.send()	Отправка ответа различных типов
res.sendFile	Отправка файла в виде потока октетов
res.sendStatus()	Установка кода состояния ответа и отправка представления в виде строки в качестве тела ответа

Самый распространенный способ отправки ответа представляет функция send(). В качестве параметра эта функция может принимать объект Buffer, объект javascript, массив или строку, в том числе с html-кодом, как в следующем примере:

```
var express = require('express');
var app = express();
app.use(function (req, res) {
    res.send("<h2>Hello</h2>");
});
app.listen(3000);
```

Пример отправки объекта:

```
res.send({id:7, name: "Dima"});
```

Пример отправки массива:

```
res.send(["Vova", "Dima", "Oleg"]);
```

Пример отправки объекта:

```
res.send(Buffer.from("Hello Express"));
```

В последнем примере объект Buffer формально представляет некоторые бинарные данные. При выполнении кода по умолчанию браузер загрузит файл, в котором будет строка «Hello Express».

Метод send удобен для отправки строк, некоторого кода html небольшой длины, однако если отправляемый код html довольно большой, то код приложения тоже становится громоздким. Правильнее размещать код html в отдельных файлах, а затем отправлять эти файлы с помощью функции sendFile().

Определим в папке проекта новый файл index.html, отправим этот файл с помощью функции sendFile:

```
var express = require('express');
var app = express();
app.use(function(req, res) {
    res.sendFile(__dirname + "/index.html");
});
app.listen(3000);
```

Следует учитывать, что в функцию sendFile необходимо передавать абсолютный путь к файлу, поэтому с помощью __dirname получаем абсолютный путь к текущему проекту и затем добавляем к нему путь к файлу в рамках текущего проекта.

Зачастую необходимо посылать клиенту статусный код ответа с некоторым сообщением по умолчанию. Функция sendStatus() позволяет это сделать. Например, отправим статусный код 404, который говорит, что ресурс не найден:

```
var express = require('express');
```

```
var app = express();
app.use("/home/foo/bar", function(req, res) {
    res.sendStatus(404)
});
app.listen(3000);
```

В случае, если нужно отправлять более информативные сообщения, можно использовать комбинацию функции status(), которая также отправляет статусный код, и функции send():

```
var express = require('express');
var app = express();
app.use("/home/foo/bar", function(req, res) {
    response.status(404).send('Ресурс не найден');
});
app.listen(3000);
```

Для переадресации применяется метод redirect():

```
redirect([status,] path)
```

В качестве параметра path передается путь, на который будет перенаправляться пользователь. Дополнительный параметр status задает статусный код переадресации. Если этот параметр не задан, тогда по умолчанию отправляется статусный код 302, который предполагает временную переадресацию.

С помощью данного метода можно выполнять переадресацию как по относительным путям, так и по абсолютным, в том числе на другие домены:

```
var express = require('express');
var app = express();
app.use("/index", function (req, res) {
    res.redirect("https://google.com")
});
app.listen(3000);
```

В данном случае при обращении по пути /index будет сделана переадресация на сайт https://google.com.

Переадресация по относительным путям также не очень сложна, но здесь важно учитывать, как именно определяется адрес для редиректа. Рассмотрим редирект относительно текущего пути, с которого производится переадресация. Например:

```
var express = require('express');
var app = express();
app.use("/home", function (req, res) {
    res.redirect("about")
});
app.use("/about", function (req, res) {
    res.send("<h1>About</h1>");
});
app.listen(3000);
```

В данном случае происходит перенаправление с ресурса /home на ресурс /about, т. е. с http://localhost:3000/home на http://localhost:3000/about.

Рассмотрим похожий пример, когда происходит переадресация с ресурса /home/bar на ресурс /home/about:

```
app.use("/home/bar", function(req, res) {
    res.redirect("about")
});
app.use("/home/about", function(req, res) {
    res.send("<h1>About</h1>");
});
```

Если необходимо выполнить переадресацию не относительно текущего ресурса, а относительно корневого каталога приложения, то в начале адреса ставится символ «/»:

```
app.use("/home/bar", function(req, res) {
    res.redirect("/about")
```

```
});
app.use("/about", function(req, res) {
    res.send("<h1>About</h1>");
});
```

Пример переадресации на два уровня выше — с http://localhost:3000/home/foo/bar на http://localhost:3000/home:

```
app.use("/home/foo/bar", function (req, res) {
    res.redirect("..")
});
```

По умолчанию при переадресации передается статусный код 302, который указывает, что ресурс временно доступен по новому адресу. Но можно указать статусный код 301, чтобы сделать переадресацию постоянной:

```
res.redirect(301, "/about");
```

Передача данных в приложение. Параметры строки запроса

Одним из способов передачи данных в приложение является использование параметров строки запроса. Можно считать, что строка запроса — это часть URL-адреса, которая идет после знака вопроса и состоит из пар «название параметра = значение», разделенных знаком амперсанда. Например, в запросе

```
http://localhost:3000/about?id=4&name=Dima
```

часть id=4&name=Dima представляет строку запроса. Здесь присутствует два параметра: параметр «id» имеет значение 4 и параметр «name» имеет значение «Dima».

В Express можно получить значение параметров строки запроса через свойство query объекта request, который передается в функцию обработки запроса. Например:

```
var express = require('express');
var app = express();
app.get("/", function(req, res){
    res.send("<h1>Главная страница</h1>");
});
app.use("/about", function(req, res){
    let id = req.query.id;
    let userName = req.query.name;
    res.send("<h1>Информация</h1><p>id=" + id +"</p>
    <p>name=" + userName + "</p>");
});
app.listen(3000);
```

С помощью выражения request.query можно получить все параметры строки запроса в виде объекта javascript, а с помощью выражения request.query.название_параметра можно обратиться к каждому отдельному параметру. Подобным образом можно передавать массивы данных:

```
var express = require('express');
var app = express();
app.get("/", function(request, response){
    response.send("<h1>Главная
        страница</h1>");
});
app.use("/about", function(request, response){
    console.log(request.query);
    let names = request.query.name;
    let resText = "<ul>";
    for(let i=0; i < names.length; i++){
        resText += "<li>" + names[i] + "</li>";
    }
    resText += "</ul>";
    response.send(resText);
});
app.listen(3000);
```

В данном случае в приложение будет передаваться параметр name, который представляет массив.

Также можно передавать более сложные объекты, которые состоят из множества свойств:

```
var express = require('express');
var app = express();
app.use("/about", function(request, response){
    console.log(request.query);
    let id = request.query.user.id;
    let name = request.query.user.name;
    response.send("<h3>id:" + id + "<br>name: " + name +"</h3>");
});
app.listen(3000);
```

В этом случае получаем объект user, который содержит два свойства — id и name, например, user: { id: '7', name: 'Tom' }. При передаче свойства объекта в строке запроса помещаются в квадратные скобки: user[id].

POST-запросы и отправка форм

При отправке сложных данных обычно используются формы. Рассмотрим, как получать отправленные данные в Express. Для получения данных форм из запроса необходимо использовать специальный пакет body-parser, поэтому вначале необходимо добавить его в проект с помощью команды:

```
npm install body-parser --save
```

Теперь определим в папке проекта новый файл register.html:

```
<! DOCTYPE html>
<html>
<head>
    <title>Регистрация</title>
    <meta charset="utf-8" />
```

```
</head>
<body>
    <h1>Введите данные</h1>
    <form action="/register" method="post">
        <label>Имя</label>
        <input type="text" name="userName" />
        <label>Возраст</label>
        <input type="number" name="userAge" />
        <input type="submit"/>
    </form>
</body>
</html>
```

Здесь определены два поля для ввода имени и возраста пользователя. После нажатия на кнопку данные будут отправлены по адресу /register.

В файле app.js определим следующий код:

```
var express = require('express');
var bodyParser = require("body-parser");
var app = express();
/* создаем парсер для данных application/x-www-form-urlencoded*/
const urlParser =
bodyParser.urlencoded({extended: false});
app.get("/register", urlParser, function
    (request, response) {
        response.sendFile(__dirname +
            "/register.html");
    });
app.post("/register", urlParser, function
    (request, response) {
        if(!request.body) return
            response.sendStatus(400);
        console.log(request.body);
        response.send(`${request.body.userName} -
        ${request.body.userAge}`);
```

```
});
app.get("/", function(request, response){
    response.send("Главная страница");
});
app.listen(3000);
```

П р и м е р. Прежде всего для получения отправленных данных необходимо создать парсер:

```
const urlParser = bodyParser.urlencoded
({extended: false});
```

Поскольку данные отправляются с помощью формы, то для создания парсера применяется функция urlencoded(). В эту функцию передается объект, устанавливающий параметры парсинга. Значение extended: false указывает, что объект — результат парсинга — будет представлять собой набор пар ключ–значение, а каждое значение может быть представлено в виде строки или массива.

Когда пользователь впервые обратится (разумеется, с использованием метода GET) по адресу /register, сработает метод app.get, который отправит пользователю файл register.html.

Когда пользователь заполнит форму, он отправит ее с помощью метода POST, а потому для обработки нужно создать функцию app.post(«/register»,…). Первый параметр функции — адрес, на который идет отправка, в данном случае /register. Второй параметр — созданный парсер, третий — функция-обработчик. Отметим, что в данном случае адресом /register связаны две функции: первая обрабатывает запросы get, а вторая — запросы post.

Для получения самих отправленных данных используем выражения типа request.body.userName, где request.body инкапсулирует данные формы, а userName — ключ данных, который соответствует значению атрибута name поля ввода на html-странице:

```
<input type="text" name="userName" />
```

Параметры маршрута

Параметры маршрутов — это именованные сегменты URL, которые используются для выбора значений из указанной позиции URL. Именованные сегменты начинаются с двоеточия, после которого следует имя (например, /: your_parameter_name/). Выбранные значения сохраняются в объекте req.params, причем имя параметра используется как ключ.

Предположим, например, что URL содержит информацию о пользователях и книгах: http://localhost:3000/users/34/books/8989. Можно извлечь эту информацию из URL и поместить в параметры userId и bookId пути:

```
app.get('/users/: userId/books/: bookId', function (req, res) {
    // доступ к userId через: req.params.userId
    // доступ к bookId через: req.params.bookId
    res.send(req.params);
})
```

Имена параметров пути должны состоять из символов (A–Z, a–z, 0–9, _). Не надо путать параметры маршрутов с параметрами строки запроса.

З а м е ч а н и е. URL /book/create будет соответствовать маршрутам вида /book/: bookId (и «create» станет значением bookId). Используется первый маршрут, соответствующий введенному URL, поэтому если необходимо обрабатывать URL вида /book/create отдельно, то обработчик этого маршрута должен быть расположен до обработчика маршрута вида /book/: bookId.

5.3. Использование промежуточных обработчиков

Работа приложения Express состоит, по сути, из серии вызовов функций промежуточной обработки. Напомним, что функции промежуточной обработки — это функции, имеющие доступ к объекту запроса, объекту ответа и к следующей функции промежуточной

обработки в цикле «запрос–ответ». Следующая функция промежуточной обработки, как правило, обозначается переменной next.

Приложение Express может использовать следующие типы промежуточных обработчиков:

1) промежуточный обработчик уровня приложения;
2) промежуточный обработчик уровня маршрутизатора;
3) промежуточный обработчик для обработки ошибок;
4) встроенные промежуточные обработчики;
5) промежуточные обработчики сторонних поставщиков программного обеспечения.

Промежуточные обработчики уровня приложения и уровня маршрутизатора можно загружать с помощью необязательного пути для монтирования. Также можно загрузить последовательность функций промежуточной обработки одновременно, в результате чего создается вспомогательный стек системы промежуточных обработчиков в точке монтирования.

Промежуточный обработчик уровня приложения — функции app.use() и app.METHOD(), где METHOD — метод HTTP-запроса (написанный в нижнем регистре), обрабатываемый функцией промежуточной обработки, например, GET, PUT или POST.

Ниже в примере представлена функция промежуточной обработки без пути монтирования. Эта функция выполняется при каждом получении запроса приложением.

```
var app = express();
app.use(function (req, res, next) {
    console.log('Time:', Date.now());
    next();
});
```

Приведем пример, где определяется последовательность функций промежуточной обработки с указанием пути монтирования[9]. Этот пример иллюстрирует создание стека промежуточных обра-

[9] Путь, который указывается в аргументе метода app.use, в Express иногда называют точкой или путем монтирования.

ботчиков, с выводом информации о запросе для всех типов запросов HTTP, адресованных ресурсам с путем /user/: id.

```
app.use('/user/: id', function(req, res, next) {
    console.log('Request URL:',
        req.originalUrl);
    next();
}, function (req, res, next) {
    console.log('Request Type:', req.method);
    next();
});
```

Express позволяет пропустить нижеописанные функции дополнительной обработки в стеке промежуточных обработчиков маршрутизатора; для этого в текущей функции-обработчике нужно вызвать next('route') и таким образом передать управление следующему маршруту. Отметим, что next('route') работает только в функциях промежуточной обработки, загруженных с помощью функций app.METHOD() или router.METHOD().

В данном примере представлен стек промежуточных обработчиков для обработки запросов GET, адресованных ресурсам в пути /user/: id. В первом стеке два обработчика, второй из которых должен быть пропущен, если идентификатор пользователя равен нулю, и при это управление передается обработчику, описанному во втором маршруте.

```
app.get('/user/: id', function (req, res, next) {
    /* if the user ID is 0, skip to the next route*/
    if (req.params.id == 0) next('route');
    /* otherwise pass the control to the next middleware function in
this stack*/
    else next(); //
}, function (req, res, next) {
    /* render a regular page*/
    res.render('regular');
});
/* handler for the /user/: id path, which renders a special page*/
```

```
app.get('/user/: id', function (req, res, next) {
    res.render('special');
});
```

Промежуточный обработчик уровня маршрутизатора работает так же, как и промежуточный обработчик уровня приложения, но он привязан к экземпляру express.Router(). Соответственно, необходимо изучить объект Router.

```
var router = express.Router();
```

Router позволяет определить дочерние подмаршруты со своими обработчиками относительно некоторого главного маршрута. Например, определим следующее приложение:

```
var express = require('express');
var app = express();

app.use("/about", function(req, res) {
    res.send("О сайте");
});

app.use("/products/create", function(req, res) {
    res.send("Добавление товара");
});
app.use("/products/: id", function(req, res) {
    res.send(`Товар ${req.params.id}`);
});
app.use("/products/", function(req, res) {
    res.send("Список товаров");
});

app.use("/", function(req, res) {
    res.send("Главная страница");
});
app.listen(3000);
```

Здесь пять маршрутов, которые обрабатываются различными обработчиками. Из них три маршрута начинаются с «/products» и относятся к некоторому функционалу по работе с товарами (просмотр списка товаров, просмотр одного товара по идентификатору и добавление товара). Объект Router позволяет связать подобный функционал в одно целое и упростить управление им. Перепишем предыдущий пример с использованием объекта Router:

```
var express = require('express');
var app = express();

// определяем Router
var productRouter = express.Router();

/* определяем маршруты и их обработчики внутри роутера*/
productRouter.use("/create", function(req, res){
    res.send("Добавление товара");
});
productRouter.use("/: id", function(req, res){
    res.send(`Товар ${req.params.id}`);
});
productRouter.use("/", function(req, res){
    res.send("Список товаров");
});
//* сопоставляем роутер с конечной точкой /products*/
app.use("/products", productRouter);

app.use("/about", function (req, res) {
    res.send("О сайте");
});

app.use("/", function (req, res) {
    res.send("Главная страница");
});
app.listen(3000);
```

Здесь определен объект productRouter, который обрабатывает все запросы по маршруту «/products». Это главный маршрут, однако он может включать подмаршрут «/» со своим обработчиком и подмаршруты «/: id» и «/create», для которых также могут быть определены свои обработчики.

Промежуточный обработчик для обработки ошибок определяется так же, как и другие функции промежуточной обработки, но с указанием не трех, а четырех аргументов в сигнатуре (err, req, res, next).

```
app.use(function(err, req, res, next) {
    console.error(err.stack);
    res.status(500).send('Something broke!');
});
```

Промежуточный обработчик для обработки ошибок должен быть определен последним, после указания всех app.use() и вызовов маршрутов. Ответы, поступающие из функции промежуточной обработки, могут иметь любой формат, например, это может быть страница сообщения об ошибке HTML, простое сообщение или строка JSON. В целях упорядочения (и для фреймворков более высокого уровня) можно определить несколько функций промежуточной обработки ошибок, точно так же, как это допускается для обычных функций промежуточной обработки. Например, для того чтобы определить обработчик ошибок для запросов, совершаемых с помощью XHR, и для остальных запросов, можно воспользоваться следующими командами:

```
var bodyParser = require('body-parser');
var methodOverride = require('method-override');

app.use(bodyParser());
app.use(methodOverride());
app.use(logErrors);
app.use(clientErrorHandler);
app.use(errorHandler);
```

В данном примере базовый код logErrors может записывать информацию о запросах и ошибках в stderr, например:

```
function logErrors(err, req, res, next) {
    console.error(err.stack);
    next(err);
}
```

Кроме того, в данном примере clientErrorHandler определен, как указано ниже; в таком случае ошибка явным образом передается далее следующему обработчику:

```
function clientErrorHandler(err, req, res, next) {
    if (req.xhr) {
        res.status(500).send({ error: 'Something failed!' });
    } else {
        next(err);
    }
}
```

Обобщающая функция errorHandler может быть реализована так:

```
function errorHandler(err, req, res, next) {
    res.status(500);
    res.render('error', { error: err });
}
```

При передаче какого-либо объекта в функцию next() (кроме строки ‘route’) Express интерпретирует текущий запрос как ошибку и пропускает все остальные функции маршрутизации и промежуточной обработки, не являющиеся функциями обработки ошибок. Для того чтобы обработать данную ошибку определенным образом, необходимо создать маршрут обработки ошибок.

Если задан обработчик ошибок с несколькими функциями обратного вызова, можно воспользоваться параметром route, чтобы перейти к следующему обработчику маршрута. Например:

```
app.get('/a_route_behind_paywall',
    function checkIfPaidSubscriber(req, res, next) {
        if(!req.user.hasPaid) {

        // continue handling this request
        next('route');
        }
    }, function getPaidContent(req, res, next) {
        PaidContent.find(function(err, doc) {
            if(err) return next(err);
        res.json(doc);
        });
    });
```

В данном примере обработчик getPaidContent будет пропущен, но выполнение всех остальных обработчиков в app для /a_route_behind_paywall будет продолжено. Вызовы next() и next(err) указывают на завершение выполнения текущего обработчика и на его состояние: next(err) пропускает все остальные обработчики в цепочке, кроме заданных для обработки ошибок, как описано выше.

В Express предусмотрен встроенный обработчик ошибок, который обрабатывает любые возможные ошибки, встречающиеся в приложении. Этот стандартный обработчик ошибок добавляется в конец стека функций промежуточной обработки.

В случае передачи ошибки в next() без обработки с помощью обработчика ошибок эта ошибка будет обработана встроенным обработчиком ошибок. Ошибка будет показана клиенту с помощью трассировки стека. Трассировка стека не включена в рабочую среду.

При написании собственных обработчиков ошибок, как правило, вызывают методы типа res.status(500). Однако надо помнить, что ранее выполненные обработчики могли уже послать http-заголовки. Попытка установить статусный код или отправить http-заголовки,

когда ответ уже потоком направляется клиенту, приведет к ошибке ERR_HTTP_HEADERS_SENT.

При добавлении нестандартного обработчика ошибок необходимо проверить, были ли уже посланы http-заголовки, и если да, то вызвать стандартные механизмы обработки ошибок:

```
function errorHandler(err, req, res, next) {
    if (res.headersSent) {
        return next(err);
    }
    res.status(500);
    res.render('error', { error: err });
}
```

Стандартный обработчик ошибок Express закрывает соединение и отклоняет запрос.

Единственной встроенной функцией промежуточной обработки в Express является express.static (root, [options]). Эта функция отвечает за предоставление статических ресурсов приложения Express. Аргумент root указывает на корневой каталог, из которого предоставляются статические ресурсы. Необязательный объект options может содержать свойства, перечисленные в табл. 3.

Таблица 3

Опции функции static

Свойство	Описание	Тип	По умолчанию
dotfiles	Опция для предоставления файлов с точкой. Возможные значения — «allow», «deny», «ignore»	Строка	«ignore»
etag	Включение или отключение генерации etag	Булевский	true
extensions	Установка альтернативных вариантов расширений файлов	Массив	[]

Окончание табл. 3

Свойство	Описание	Тип	По умолчанию
index	Отправка файла индекса каталога. Установите значение false, чтобы отключить индексацию каталога	Смешанный	«index.html»
lastModified	Установка в заголовке Last-Modified даты последнего изменения файла в операционной системе. Возможные значения: true или false	Булевский	true
maxAge	Установка значения свойства max-age в заголовке Cache-Control, в миллисекундах, или в виде строки в формате ms	Число	0
redirect	Перенаправление к заключительному символу «/», если имя пути — каталог	Булевский	true
setHeaders	Функция для установки заголовков HTTP, предоставляемых с файлом	Функция	

Приведем пример использования функции промежуточной обработки express.static с объектом дополнительных опций:

```
var options = {
    dotfiles: 'ignore',
    etag: false,
    extensions: ['htm', 'html'],
    index: false,
    maxAge: '1d',
    redirect: false,
    setHeaders: function (res, path, stat) {
```

```
        res.set('x-timestamp', Date.now());
    }
}

app.use(express.static('public', options));
```

Для каждого приложения допускается наличие нескольких статических каталогов:

```
app.use(express.static('public'));
app.use(express.static('uploads'));
app.use(express.static('files'));
```

Для использования промежуточных обработчиков сторонних поставщиков программного обеспечения необходимо установить модуль Node.js для соответствующей функциональной возможности, затем загрузить его на уровне приложения или на уровне маршрутизатора.

Для расширения функциональности приложений Express используются промежуточные обработчики сторонних поставщиков программного обеспечения. Необходимо установить модуль Node.js для соответствующей функциональной возможности, затем загрузить его на уровне приложения или на уровне маршрутизатора. В приведенном ниже примере показана установка и загрузка функции промежуточной обработки для синтаксического анализа cookie — cookie-parser.

```
$ npm install cookie-parser

var express = require('express');
var app = express();
var cookieParser = require('cookie-parser');

// load the cookie-parsing middleware
app.use(cookieParser());
```

Промежуточные обработчики уровня приложения и уровня маршрутизатора можно загружать с помощью необязательного пути для монтирования. Также можно загрузить последовательность функций промежуточной обработки одновременно, в результате чего создается вспомогательный стек системы промежуточных обработчиков в точке монтирования.

5.4. JSON и AJAX

JSON — один из самых популярных форматов хранения и передачи данных, и Express имеет все возможности для работы с данными, представленными в этом формате.

Пусть в папке проекта имеется файл index.html со следующим кодом:

```
<! DOCTYPE html>
<html>
<head>
    <title>Регистрация</title>
    <meta charset="utf-8" />
</head>
<body>
        <h1>Введите данные</h1>
        <form name="registerForm">
        <label>Имя</label></br>
        <input type="text" name="userName" />
        <label>Возраст</label></br>
        <input type="number" name="userAge" />
        <button type="submit" id="submit">
        Отправить</button>
    </form>
    <script>
document.getElementById("submit").
    addEventListener("click", function (e) {
    e.preventDefault();
```

```
    // получаем данные формы
    let registerForm =
        document.forms["registerForm"];
    let userName =
        registerForm.elements["userName"].value;
    let userAge =
        registerForm.elements["userAge"].value;
    // сериализуем данные в json
    let user = JSON.stringify({userName:
userName, userAge: userAge});
    let request = new XMLHttpRequest();
    // посылаем запрос на адрес "/user"
    request.open("POST", "/user", true);
    request.setRequestHeader("Content-Type", "application/json");
    request.addEventListener("load", function(){
        console.log(request.response);
        // смотрим ответ сервера
    });
        request.send(user);
});
    </script>
</body>
<html>
```

Здесь создана форма с двумя полями для ввода имени и возраста пользователя. В обработчике клика отменяется отправка этой формы. Далее значения полей формы сериализуются в объект JSON, который затем отсылается с помощью AJAX на сервер по адресу /user.

Создадим главный файл приложения app.js, который принимает отправленные данные и обрабатывает их.

```
const express = require("express");
const app = express();
// создаем парсер для данных в формате json
const jsonParser = express.json();
```

```
app.post("/user", jsonParser, function
(request, response) {
console.log(request.body);
if(!request.body) return
response.sendStatus(400);
console.log(request.body);
response.json(`${request.body.userName} -
${request.body.userAge}`);
});

app.get("/", function(request, response){
response.sendFile(__dirname +
"/index.html");
});

app.listen(3000);
```

Разберем пример построчно. Прежде всего для получения данных в формате JSON необходимо создать парсер с помощью функции json:

```
const jsonParser = express.json();
```

В реальности этот парсер будет использовать модуль bodyParser, который применялся нами ранее для парсинга данных отправленной формы. Для получения данных, как и в случае с формами, используются выражения типа request.body.userName, где request.body инкапсулирует данные формы, а userName — ключ данных. Поскольку взаимодействие с клиентом происходит через формат JSON, то данные клиенту отправляются с помощью метода response.json().

При обращении к корню веб-приложения пользователю будет отправляться содержимое файла index.html с формой ввода данных. Для тестирования приложения надо его запустить и обратиться к корню веб-сайта. Далее надо ввести какие-либо данные, и после отправки формы в консоли браузера отобразится ответ сервера.

5.5. Шаблонизаторы

Распространенным паттерном веб-проектирования является MVC (англ. *Model-View-Controller*, модель-представление-контроллер), суть которого состоит в разделении данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер. Модель предоставляет данные и реагирует на команды контроллера, изменяя свое состояние. Представление отвечает за отображение данных модели пользователю. Контроллер интерпретирует действия пользователя, оповещая модель о необходимости изменений.

Представление — это отдельный программный компонент веб-приложения. Представление воплощено (англ. *implement*) в виде одного или нескольких js- или hbs-файлов и т. п. Приложение можно реализовать в виде набора файлов-шаблонов и обрабатывающего их шаблонизатора. При этом шаблоны разработчики обычно создают самостоятельно, учитывая в них специфический дизайн приложения, а шаблонизатор используют стандартный — с помощью менеджера пакетов скачивают с удаленного репозитория и инсталируют в папку проекта.

В простом случае шаблоны — это файлы, содержащие html-разметку и инструкции для шаблонизатора (например, такие инструкции: «вставь на это место определенные данные», «в случае отсутствия данных не отображай эту часть веб-страницы» и др.).

Одним из наиболее популярных, быстрых и многофункциональных шаблонизаторов для JavaScript является Handlebars. Он принимает на вход любую строку, состоящую из HTML-тегов и специальных выражений, и компилирует ее в функцию JavaScript. Эта функция, в свою очередь, принимает один параметр — объект данных и возвращает строку HTML, где определенные свойства объекта уже вставлены в нужные места шаблона.

Handlebars практически не позволяет добавлять логику и произвольный JavaScript в шаблоны (кроме циклов и условных выражений, о которых будет сказано ниже), тем самым он как бы «заставляет» разработчиков содержать логику отдельно от разметки.

Handlebars используют такие JavaScript-фреймворки, как Meteor.js, Derby.js, Ember.js, также с ним отлично взаимодействуют и другие фреймворки, например Backbone.js.

Изучим работу Handlebars на примерах. Для работы с представлениями необходимо установить пакет hbs в проект:

```
npm install hbs --save
```

Для хранения представлений определим в проекте папку views. Затем в нее добавим новый файл mycontact.hbs, где hbs — это расширение по умолчанию для представлений, которые обрабатываются движком Handlebars. В файле contact.hbs определим простейший html-код:

```
<! DOCTYPE html>
<html>
<head>
    <title>Контакты</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>Контакты</h1>
    <p>Электронный адрес: admin@mycorp.com</p>
</body>
<html>
```

На данном этапе представление не содержит никаких шаблонов. Изменим файл приложения app.js. Чтобы установить Handlebars в качестве движка представлений в Express, вызывается функция:

```
app.set("view engine", "hbs");
```

Для маршрута «/contact» определим функцию обработчика, производящую рендеринг представления «contact.hbs» с помощью функции res.render(), которая на основе представления создает страницу html и отправляет ее клиенту:

```
app.use("/contact", function(req, res){
    res.render("contact.hbs");
});
```

В представление можно передавать данные из модели, которые движок представлений использует для рендеринга. Изменим файл app.js следующим образом:

```
app.use("/contact", function(req, res){
    res.render("mycontact.hbs", {
        title: "Мои контакты",
        email: "name@mycorp.com",
        phone: "+12345678910"
    });
});
```

Теперь в качестве второго параметра в функцию res.render() передается специальный объект с тремя свойствами.

Далее изменим код представления mycontact.hbs:

```
<! DOCTYPE html>
<html>
<head>
    <title>{{title}}</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>{{title}}</h1>
    <p>Электронный адрес: {{email}}</p>
    <p>Телефон: {{phone}}</p>
</body>
<html>
```

Вместо конкретных данных в коде представления используются те данные, которые определены в модели. Чтобы обратиться к свойствам модели, в двойных фигурных скобках указывается нужное

свойство: {{title}}. При рендеринге представления вместо подобных выражений будут вставляться значения соответствующих свойств модели.

Рассмотрим более сложный случай. Пусть в представление передается массив:

```
app.use("/contact", function(req, res){
    res.render("mycontact.hbs", {
        title: "Мои контакты",
        emailsVisible: true,
        emails: ["name@mycorp.com",
            "name2@mycorp.com"],
        phone: "+12345678910"
    });
});
```

Для вывода данных изменим представление mycontact.hbs:

```
<body>
    <h1>{{title}}</h1>
    {{#if emailsVisible}}
    <h3> Электронные адреса</h3>
    <ul>
        {{#each emails}}
        <li> {{this}} </li>
        {{/each}}
    </ul>
    {{/if}}
    <p>Телефон: {{phone}} </p>
</body>
```

Здесь выражение типа

```
{{#if emailsVisible}}
// код
{{/if}}
```

позволяет определить видимость кода в зависимости от значения свойства emailsVisible: если это свойство равно true, то блок кода между {{#if emailsVisible}} и {{/if}} добавляется на веб-страницу.

Для перебора массивов можно воспользоваться конструкцией each:

```
{{#each emails}}
    <li>{{this}}</li>
{{/each}}
```

Эта конструкция перебирает все элементы из массива emails и создает для них элемент <li>. Текущий перебираемый элемент помещается в переменную this. В итоге при обращении по пути «/contact» на веб-странице в виде списка будет отображаться массив.

По умолчанию представления помещаются в папку views, но можно выбрать любую другую папку в проекте. Для этого необходимо установить параметр views. В данном случае в качестве папки представлений используется папка temp.

```
const express = require("express");
const app = express();

app.set("view engine", "hbs");
app.set("views", "temp");
// установка пути к представлениям
app.use("/contact", function(req, res){
    res.render("contact");
});
app.listen(3000);
```

Нередко на веб-страницах в приложении используются какие-то общие элементы. Это может быть меню, шапка сайта, футер, другие элементы. Однако здесь возникает проблема: если потребуется поменять этот общий элемент, то придется вносить изменения на все веб-страницы, которые его используют. Разумеется, гораздо проще определить этот элемент в одном месте, а затем подключать на все

страницы. Решить эту проблему помогают частичные представления (англ. *partial views*), позволяющие вставлять общие разделяемые элементы в обычные представления. К примеру, можно сделать общее меню и общий футер. Для этого создадим для частичных представлений в проекте подкаталог views/partials.

Добавим в папку views/partials новый файл menu.hbs:

```
<nav>
    <a href="/">Главная</a> |
    <a href="/contact">Контакты</a>
</nav>
```

Затем также добавим в views/partials новый файл footer.hbs:

```
<footer><p> Copyright 2018</p></footer>
```

Это два частичных представления, которые будут вставлены в обычные представления. В папке views определим обычное представление contact.hbs:

```
<body>
    {{> menu}}

    <h1>{{title}}</h1>
    <p>Электронный адрес: {{email}}</p>
    <p>Телефон: {{phone}}</p>

    {{> footer}}
</body>
```

Для вставки частичного представления применяется выражение {{> menu}}, в котором прописывается имя файла частичного представления без расширения.

Также добавим в папку views новое представление, которое назовем home.hbs:

```
<body>
    {{> menu}}
```

```
    <h1>Главная страница</h1>

    {{> footer}}
</body>
```

Таким образом, есть два представления, которые имеют общие элементы, и если потребуется добавить какой-нибудь новый пункт меню, то достаточно изменить файл menu.hbs.

В итоге структура проекта будет следующая:

```
app.js
node_modules
views
    contact.hbs
    home.hbs
    partials
        footer.hbs
        menu.hbs
```

Изменим файл app.js:

```
/*для настройки функционала частичных представлений*/
hbs.registerPartials(__dirname + "/views/partials");
app.use("/contact", function(req, res){
    res.render("contact", {
        title: "Мои контакты",
        emailsVisible: true,
        emails: ["name@mycorp.com",
            "name1@mycorp.com"],
        phone: "+12345678910"
    });
});
```

Если запустить проект и обратиться по одному из двух маршрутов: «/» или «/contact», то после рендеринга представления веб-страница будет содержать меню и футер.

Еще одним важным способом работы с шаблонами является использование мастер-страницы или файла layout, который позволяет определить общий макет всех веб-страниц сайта. Благодаря этому гораздо проще обновлять сайт, определять и менять какие-то общие блоки кода.

Для работы с файлами layout установим в проект модуль express-handlebars с помощью следующей команды:

```
npm install express-handlebars
```

Пусть в проекте в папке views/partials будут определены два частичных представления для меню и футера: первое представление — menu.hbs, второе представление — footer.hbs. Создадим в проекте в папке views новый каталог layouts и определим в нем файл layout.hbs, который будет определять макет сайта:

```
<! DOCTYPE html>
<html>
<head>
    <title>{{title}}</title>
    <meta charset="utf-8" />
</head>
<body>
    {{> menu}}

    {{{body}}}

    {{> footer}}
</body>
<html>
```

Здесь внедряются частичные представления menu.hbs и footer.hbs. И кроме того, здесь также присутствует такое выражение, как {{{body}}}. Вместо этого выражения будем вставлять содержимое конкретных представлений.

Затем в папке views определим два обычных представления. Представление contact.hbs:

```
<h1>{{title}}</h1>
<p>Электронный адрес: {{email}}</p>
<p>Телефон: {{phone}}</p>
```

и представление home.hbs:

```
<h1>Главная страница</h1>
```

Эти представления не содержат элементов body, head, каких-то общих блоков, так как все они определены в файле layout.hbs.

Далее определим следующий файл — app.js:

```
const express = require("express");
const expressHbs = require("express-handlebars");
const hbs = require("hbs");
const app = express();

// устанавливаем настройки для файлов layout
app.engine("hbs", expressHbs(
    {
        layoutsDir: "views/layouts",
        defaultLayout: "layout",
        extname: "hbs"
    }
))
app.set("view engine", "hbs");
hbs.registerPartials(__dirname +
    "/views/partials");

app.use("/contact", function(req, res){
    res.render("contact", {
        title: "Мои контакты",
        emailsVisible: true,
```

```
        emails: ["name@mycorp.com",
            "name1@mycorp.com"],
        phone: "+12345678910"
    });
});

app.use("/", function(req, res)
    res.render("home.hbs");
});
app.listen(3000);
```

Функция expressHbs осуществляет конфигурацию движка. В частности, свойство layoutsDir задает путь к папке с файлами layout относительно корня каталога проекта. Свойство defaultLayout указывает на название файла, который будет использоваться в качестве мастер-страницы. В данном случае это файл layout.hbs, поэтому указываем название этого файла без расширения. Третье свойство extname задает расширение файлов.

Далее будем рассматривать хелперы. Хелперы — это функции, возвращающие некоторую строку, которую можно добавить в любое место представления. Строка может представлять собой в том числе и код html.

Хелперы позволяют оптимизировать создание кода представлений. В частности, можно один раз определить функцию хелпера, а затем многократно применять ее в самых различных местах для генерации кода.

Для применения хелперов изменим код файла app.js:

```
const express = require("express");
const hbs = require("hbs");
const app = express();
hbs.registerHelper("getTime", function(){
    var myDate = new Date();
    var hour = myDate.getHours();
    var minute = myDate.getMinutes();
    var second = myDate.getSeconds();
```

```
    if (minute < 10) {
        minute = "0" + minute;
    }
    if (second < 10) {
        second = "0" + second;
    }
    return "Текущее время: " + hour + ":" +
        minute + ":" + second;
});
app.set("view engine", "hbs");
app.get("/", function(req, res){
        res.render("home.hbs");
});
app.listen(3000);
```

Хелпер определяется с помощью функции hbs.registerHelper(). Первый параметр функции — название хелпера, а второй — функция, которая возвращает строку. В данном случае возвращается текущее время.

Далее определим представление home.hbs:

```
<! DOCTYPE html>
<html>
<head>
    <title>Главная страница</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>Главная страница</h1>
    <div>{{getTime}}</div>
</body>
<html>
```

Для вызова хелпера в двойных фигурных скобках указывается его имя.

Хелпер может возвращать не только литеральную строку, но и код html. Кроме того, хелперу можно передавать параметры, которые применяются при генерации результата. Например, определим в app.js еще один хелпер:

```
hbs.registerHelper("createStringList",
function(array){
    var result="";
    for(var i=0; i<array.length; i++){
        result +="<li>" + array[i] + "</li>";
    }
    return new hbs.SafeString("<ul>" + result +
        "</ul>");
});
app.get("/", function(req, res){
    response.render("home.hbs", {
        fruit: ["apple", "lemon", "banana"]
    });
});
```

Здесь добавлено определение хелпера createStringList(), который в качестве параметра принимает некоторый массив строк и из них создает элемент <ul>. Однако чтобы возвращаемое значение расценивалось именно как html, его надо обернуть в функцию hbs.SafeString().

Также изменим файл представления home.hbs:

```
<! DOCTYPE html>
<html>
<head>
    <title>Главная страница</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>Главная страница</h1>
    <div>{{getTime}}</div>
```

```
    <div>{{createStringList fruit}}</div>
</body>
<html>
```

В итоге на веб-страницу будет выведен список. При этом, определив хелпер один раз, можно его использовать многократно в различных представлениях, передавая ему различные значения.

Контрольные вопросы

1. Как запустить Express.js app в режиме кластера? Какие преимущества дает такой вариант запуска?

2. Может ли функция-обработчик запроса быть асинхронной? Иными словами, можно ли в Express.js использовать в обработчиках запросов async/await?

3. В чем разница между режимами «development» и «production» в Express.js?

Глава 6
ИНТЕГРАЦИЯ ФРЕЙМВОРКА EXPRESS И MONGODB

6.1. Начало работы с MongoDB

Одной из наиболее популярных систем управления базами данных (СУБД) для Node.js на данный момент является MongoDB. Это документоориентированная СУБД с открытым исходным кодом, не требующая описания схемы таблиц. Классифицируется как NoSQL, использует JSON-подобные документы и схему базы данных.

Перечислим основные особенности системы MongoDB: поддержка ad-hoc-запросов, которые могут возвращать конкретные поля документов и пользовательские JavaScript-функции; поддержка поиска по регулярным выражениям; настройка запросов на возвращение случайного набора результатов; поддержка индексов.

Система может работать с набором реплик, т. е. содержать две или более копии данных на различных узлах. Каждый экземпляр набора реплик может в любой момент выступить в роли основной или вспомогательной реплики. Все операции записи и чтения по умолчанию осуществляются с основной репликой. Вспомогательные реплики поддерживают в актуальном состоянии копии данных. В случае, когда основная реплика дает сбой, набор реплик проводит выбор и определяет, какая реплика должна стать основной. Второстепенные реплики могут дополнительно являться источником для операций чтения.

Система масштабируется горизонтально, используя технику сегментирования (англ. *sharding*) объектов баз данных — распределение их частей по различным узлам кластера. Администратор выбирает ключ сегментирования, который определяет, по какому критерию данные будут разнесены по узлам (в зависимости от значений хэша ключа сегментирования). Благодаря тому, что каждый узел кластера может принимать запросы, обеспечивается балансировка нагрузки.

MongoDB реализует подход к построению баз данных (БД), где нет таблиц, схем, запросов SQL, внешних ключей и многих других концепций, присущих объектно-реляционным БД. В отличие от реляционных БД, MongoDB предлагает документоориентированную модель данных, благодаря чему MongoDB в некоторых случаях работает быстрее и обладает лучшей масштабируемостью.

Одним из популярных стандартов обмена данными и их хранения является стандарт JSON, эффективно описывающий сложные по структуре данные. Способ хранения данных в MongoDB похож на формат JSON, хотя формально JSON в этой системе не используется. Для хранения в MongoDB применяется формат, который называется BSON (сокращение от binary JSON).

Данные, хранящиеся в BSON, занимают больше места, чем данные в JSON-формате, однако этот недостаток окупается скоростью работы. BSON позволяет быстрее выполнять поиск и обработку данных.

Некоторым аналогом таблиц, существующих в реляционных БД, в MongoDB являются коллекции. В реляционных БД таблицы хранят однотипные жестко структурированные объекты, а коллекции могут содержать разные объекты, имеющие различную структуру и различный набор свойств.

В таблицах реляционных БД хранятся строки, а в MongoDB — документы. В отличие от строк документы могут хранить сложную по структуре информацию. Документ можно представить как хранилище ключей и значений.

Для работы с MongoDB необходимо прежде всего установить сервер MongoDB, скачав архив с официального сайта и распаковав его, например, в папку C:\mongodb. В папке C:\mongodb\bin нахо-

дится много приложений, которые выполняют определенную роль. Среди них есть:

1) mongod — сервер баз данных MongoDB, который обрабатывает запросы, управляет форматом данных и выполняет различные операции в фоновом режиме по управлению базами данных;

2) mongo — консольный клиент для взаимодействия с базами данных.

Запустим сервер БД, выполнив к консоли команду

```
C:\mongodb\bin\mongod
```

В другом окне запустим клиента и поизведем простейшие операции для примера

```
C:\mongodb\bin\mongo
>use test
>db.users.save({ name: "Dima" })
>db.users.find()
```

Команда use test устанавливает в качестве используемой базу данных test. Если такой БД нет, то она создается автоматически, и далее объект db будет представлять текущую базу данных. Свойство users — это коллекция, в которую затем мы добавляем новый объект. Если в SQL нам надо создавать таблицы заранее, то MongoDB создает коллекции самостоятельно при их отсутствии.

С помощью метода db.users.save() в коллекцию users базы данных test добавляется объект { name: "Dima" }. Если объект был успешно добавлен, то консоль выведет результ в виде выражения WriteResult({ "nInserted" : 1 }). Последняя команда db.users.find() выводит на экран все объекты из БД test.

На этом примере виден шаблон взаимодействия с сервером БД в MongoDB. Можно выделить следующие этапы этого взаимодействия:

1) подключение к серверу;

2) получение объекта базы данных на сервере;

3) получение объекта коллекции в базе данных;

4) взаимодействие с коллекцией (добавление, удаление, получение, изменение данных).

6.2. Взаимодействие с MongoDB из Node.js

Взаимодействовать с сервером MongoDB можно не только из стандартного клиента, но и из Node.js, для этого, разумеется, необходим соответствующий драйвер.

Создадим новый проект. Для этого создадим новый каталог, который будет называться mongoapp, в нем — новый файл package.json:

```
{
    "name": "mongoapp",
    "version": "1.0.0",
    "dependencies": {
        "express": "^4.16.0",
        "body-parser": "^1.18.0",
        "mongodb": "^3.1.0"
    }
}
```

Далее перейдем в этот каталог и в командной строке для добавления всех нужных пакетов выполним команду

```
npm install
```

Рассмотрим пример работы с сервером MongoDB.

```
const MongoClient =
    require("mongodb").MongoClient;

const url = "mongodb://localhost:27017/";
const mongoClient = new MongoClient(url,
    { useNewUrlParser: true });
```

```
mongoClient.connect(function(err, client){
    const db = client.db("usersdb");
    const collect = db.collection("users");
    let user = {name: "Tom", age: 23};
    collect.insertOne(user, function(err, res){
        if(err){
            return console.log(err);
        }
        console.log(res.ops);
        client.close();
    });
});
```

Ключевым классом для работы с MongoDB является класс MongoClient, через него осуществляется все взаимодействие с хранилищем данных. Соответственно вначале необходимо импортировать MongoClient.

Затем создается объект mongoClient. Для этого в его конструктор передается два параметра. Первый параметр — URL сервера БД. В качестве протокола используется «mongodb://», localhost соответствует локальной машине, номер порта по умолчанию равен 27017.

Второй параметр — это необязательный объект конфигурации, который в данном случае указывает инфраструктуре mongodb на необходимость использования нового парсера адреса сервера.

Далее с помощью метода connect происходит подключение к серверу. В качестве параметра метод принимает функцию обратного вызова, которая срабатывает при установке подключения. Эта функция принимает два параметра: err (возникшая ошибка при установке соединения) и client (ссылка на подключенного к серверу клиента).

Используя объект подлюченного клиента, мы обращаемся к БД на сервере — в данном примере это «usersdb». Если на сервере MongoDB нет этой БД, то при первом обращении к ней сервер автоматически ее создаст. После подключения мы обращаемся к коллекции «users». И снова, если такой коллекции нет в БД usersdb, она будет создана автоматически.

Получив коллекцию, можно использовать ее методы. В данном случае мы добавляем один документ — объект user. Для этого используется метод insertOne(), который имеет два параметра — добавляемый объект и функцию обратного вызова, которая выполняется после добавления. В этой функции два параметра: err (ошибка, которая может возникнуть при операции) и result (результат операции — добавленный объект).

В функции обратного вызова инспектируется добавленный объект с помощью свойства result.ops. Отметим, что это не объект user, а объект, который получен из БД и который в дополнение к свойствам name и age имеет идентификатор, установленный при добавлении.

В завершении работы с БД нам надо закрыть соединение с помощью метода client.close().

Перейдем к рассмотрению более сложного примера. Объединим в одном приложении обработку запросов с помощью Express и работу с данными в MongoDB. Для этого определим следующий файл приложения app.js:

```
const express = require("express");
const MongoClient =
    require("mongodb").MongoClient;
const objectId = require("mongodb").ObjectID;
const app = express();
const jsonParser = express.json();

const mongoClient = new MongoClient("mongodb:
    //localhost:27017/",
    { useNewUrlParser: true });

let dbClient;
app.use(express.static(__dirname + "/public"));
mongoClient.connect(function(err, client){
    if(err) return console.log(err);
    dbClient = client;
    app.locals.collection =
```

```
        client.db("usersdb").collection("users");
    app.listen(3000, function(){
        console.log("Сервер ожидает подключения…");
    });
});

app.get("/api/users", function(req, res){
    const collection =
        req.app.locals.collection;
    collection.find({}).toArray(function(err,
    users){
        if(err) return console.log(err);
        res.send(users)
    });
});

app.get("/api/users/: id", function(req, res){
    const id = new objectId(req.params.id);
    const collection =
        req.app.locals.collection;
    collection.findOne({_id: id},
        function(err, user){
        if(err) return console.log(err);
        res.send(user);
    });
});

app.post("/api/users", jsonParser,
    function (req, res) {
    if(!req.body) return res.sendStatus(400);
    const userName = req.body.name;
    const userAge = req.body.age;
    const user = {name: userName, age: userAge};
    const collection =
        req.app.locals.collection;
    collection.insertOne(user,
```

```
    function(err, result){
        if(err) return console.log(err);
        res.send(user);
    });
});

app.delete("/api/users/: id",
    function(req, res){
    const id = new objectId(req.params.id);
    const collection =
        req.app.locals.collection;
    collection.findOneAndDelete({_id: id},
        function(err, result){
        if(err) return console.log(err);
        let user = result.value;
        res.send(user);
        });
});

app.put("/api/users", jsonParser,
    function(req, res){
    if(!req.body) return res.sendStatus(400);
    const id = new objectId(req.body.id);
    const userName = req.body.name;
    const userAge = req.body.age;
    const collection =
        req.app.locals.collection;
    collection.findOneAndUpdate({_id: id},
        { $set: {age: userAge, name: userName}},
    {returnOriginal: false },
    function(err, res){
        if(err) return console.log(err);
        const user = result.value;
        res.send(user);
        });
});
```

```
// прослушиваем прерывание работы программы
process.on("SIGINT", () => {
    dbClient.close();
    process.exit();
});
```

Для каждого типа запросов здесь определен свой обработчик Express. И в каждом из обработчиков мы обращаемся к базе данных. Чтобы не открывать и не закрывать подключение при каждом запросе, мы открываем подключение в самом начале и только после открытия подключения запускаем прослушивание входящих запросов.

Поскольку все взаимодействие будет происходить с коллекцией users, то получаем ссылку на эту коллекцию в локальную переменную приложения app.locals.collection. Затем через эту переменную мы сможем получить доступ к коллекции в любом месте приложения.

В конце работы скрипта нужно закрыть подключение, сохраненное в переменной dbClient. В данном случае мы прослушиваем событие SIGINT, которое генерируется при нажатии комбинации CTRL+C в консоли.

Когда приходит GET-запрос к приложению, то мы возвращаем клиенту все документы из базы данных. Если в GET-запросе передается параметр id, то возвращаем из БД только одного пользователя, соответствующего этому параметру.

Когда приходит POST-запрос, с помощью парсера jsonParser получаем отправленные данные и по ним создаем объект, который добавляем в базу данных посредством метода insertOne().

Клиентская часть этого приложения пишется очевидным образом, а потому ее код здесь не приводится.

Контрольные вопросы

1. Что такое NoSQL?
2. Какие существуют типы хранилищ данных в NoSQL?
3. Перечислите основные преимущества MongoDB.

4. Перечислите основные недостатки MongoDB.

5. Что такое пространство имен в MongoDB?

6. Поддерживает ли MongoDB ограничения внешнего ключа (англ. *foreign key*)?

7. Можно ли использовать MongoDB в ситуациях, которые требуют атомарности обновления нескольких документов или согласованности между операциями чтения из нескольких документов? Поддерживает ли MongoDB транзакции?

8. Что такое шардинг и репликация в MongoDB? В чем отличие этих понятий?

9. Какие существуют ограничения на размер документа в MongoDB? Как спецификация GridFS позволяет обходить эти ограничения?

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ

1. ECMAScript Language Specification : [сайт]. URL: http://www.ecma-international.org/ecma-262/9.0/index.html#Title (дата обращения: 14.09.2019).

2. Cross-Origin Resource Sharing // w3.org : [сайт]. URL: https://www.w3.org/TR/cors/ (дата обращения: 14.09.2019).

3. Cross-Origin Resource Sharing // developer.mozilla.org : [сайт]. URL: https://developer.mozilla.org/ru/docs/Web/HTTP/CORS (дата обращения: 14.09.2019).

4. Express 4.17.1 Fast, unopinionated, minimalist web framework for Node.js // expressjs.com : [сайт]. URL: https://expressjs.com/ (дата обращения: 14.09.2019).

5. Веб-фреймворк Express (Node.js/JavaScript) [Электронный ресурс]. URL: https://developer.mozilla.org/ru/docs/Learn/Server-side/Express_Nodejs (дата обращения: 14.09.2019).

6. Раздел веб-разработки на METANIT.COM // METANIT.COM : [сайт о программировании]. URL: https://metanit.com/web/ (дата обращения: 14.09.2019).

Учебное издание

Солодушкин Святослав Игоревич
Юманова Ирина Фарисовна

РАЗРАБОТКА ПРОГРАММНЫХ КОМПЛЕКСОВ НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

Заведующий редакцией *М. А. Овечкина*
Редактор *Н. В. Чапаева*
Корректор *Н. В. Чапаева*
Компьютерная верстка *В. К. Матвеев*

Подписано в печать 18.06.2020 г. Формат 60 × 84 1/16.
Бумага офсетная. Цифровая печать. Усл. печ. л. 7,67.
Уч.-изд. л. 6,5. Тираж 100 экз. Заказ 115.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620083, Екатеринбург, ул. Тургенева, 4
Тел.: +7 (343) 389-94-79, 350-43-28
E-mail: rio.marina.ovechkina@mail.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620083, Екатеринбург, ул. Тургенева, 4
Тел.: +7 (343) 358-93-06, 350-58-20, 350-90-13
Факс: +7 (343) 358-93-06
http://print.urfu.ru

СОЛОДУШКИН СВЯТОСЛАВ ИГОРЕВИЧ

Кандидат физико-математических наук, доцент кафедры вычислительной математики и компьютерных наук Уральского федерального университета. Окончил математико-механический факультет Уральского государственного университета по специальности «Математика и компьютерные науки» (2004). Читает лекции по курсам «Протоколы Интернета», «Web и DHTML», «Информационные системы и сервисы», «Параллельные численные методы». Автор более 65 научных публикаций. Область научных интересов — программирование для Интернета, веб-разработка.

ЮМАНОВА ИРИНА ФАРИСОВНА

Кандидат физико-математических наук, доцент кафедры вычислительной математики и компьютерных наук Уральского федерального университета. Окончила факультет прикладной математики Ижевского государственного технического университета по специальности «Прикладная математика» (2010). Читает лекции по курсам «Протоколы Интернета», «Web и DHTML». Автор более 45 научных публикаций. Область научных интересов — математическое моделирование, численные методы и комплексы программ.